

AD-A253 740



1

Selective Perception for Robot Driving

Douglas A. Reece

May 1992
CMU-CS-92-139

DTIC
ELECTE
AUG 12 1992
S A D

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

© Douglas A. Reece, May 1992

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy*

This document has been approved
for public release and sale; its
distribution is unlimited.

This research was supported in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

92-20940



92 8 3 032

Keywords: Vision and scene understanding, active vision, robotics, knowledge representation, reasoning with uncertainty, driving, traffic simulation, tree search strategies, graphics applications

**Carnegie
Mellon**

School of Computer Science

**DOCTORAL THESIS
in the field of
Computer Science**

Selective Perception for Robot Driving

DOUGLAS REECE

**Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy**

Accession For	
NTIS	CRA&I <input checked="" type="checkbox"/>
DTIC	TAB <input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability	
Dist	Availability for special
A-1	

Statement A per telecon Chahira Hopper
WL/AAAT
WPAFB, OH 45433

ACCEPTED:

NWW 8/10/92

Steven A. Hopper

MAJOR PROFESSOR

15 May 92

DATE

IRY

DEAN

19 May 1992

DATE

APPROVED:

Paul Chahira

PROVOST

21 May 1992

DATE

DTIC QUALITY INSPECTED

Abstract

Robots performing complex tasks in rich environments need very good perception modules in order to understand their situation and choose the best action. Robot planning systems have typically assumed that perception was so good that it could refresh the entire world model whenever the planning system needed it, or whenever anything in the world changed. Unfortunately, this assumption is completely unrealistic in many real-world domains because perception is far too difficult. Robots in these domains cannot use the traditional planner paradigm, but instead need a new system design that integrates reasoning with perception. In this thesis I describe how reasoning can be integrated with perception, how task knowledge can be used to select perceptual targets, and how this selection dramatically reduces the computational cost of perception.

The domain addressed in this thesis is driving in traffic. I have developed a microscopic traffic simulator called PHAROS that defines the street environment for this research. PHAROS contains detailed representations of streets, markings, signs, signals, and cars. It can simulate perception and implement commands for a vehicle controlled by a separate program. I have also developed a computational model of driving called Ulysses that defines the driving task. The model describes how various traffic objects in the world determine what actions that a robot must take. These tools allowed me to implement robot driving programs that request sensing actions in PHAROS, reason about right-of-way and other traffic laws, and then command acceleration and lane changing actions to control a simulated vehicle.

In the thesis I develop three selective perception techniques and implement them in three robot driving programs of increasing sophistication. The first, Ulysses-1, uses *perceptual routines* to control visual search in the scene. These task-specific routines use known objects to guide the search for others—e.g. a routine scans *along the right side of the road ahead* for a sign. The second program, Ulysses-2, decides which objects are the most critical in the current situation and looks for them. It ignores objects that cannot affect the robot's actions. Ulysses-2 creates an inference tree to determine the effect of uncertain input data on action choices, and searches this tree to decide which data to sense. Finally, Ulysses-3 uses domain knowledge to reason about how dynamic objects will move or change over time. Objects that do not move enough to affect the robot can be ignored by perception. The program uses the inference tree from Ulysses-2 and a time-stamped, persistent world model to decide what to look for. When run in the PHAROS world, the techniques included in Ulysses-3 reduced the computational cost for perception by 9 to 12 orders of magnitude when compared to an uncontrolled, general perception system.

Acknowledgements

I would like to acknowledge the guidance and support that I received from various people during my journey toward a PhD. First on the list is my advisor, Steve Shafer. Steve is the person who started me off on the topic of driving in traffic, even though it seemed a little removed from the normal sort of robotics research our group conducted. He helped to guide me towards the the important issues that driving raised for robot perception. He also helped me to learn what research and writing papers was all about.

Secondly I would like to thank the people who gave me insights during informal technical discussions: Jans Aasman for long exchanges about driving (human and machine) and architectures for machine intelligence; Chuck Thorpe for practical points of view on intelligent robot control systems; Allen Newell, Tom Mitchell, Jaime Carbonell and their students in the Integrated Architectures class; and many others along the way.

Various people help make life as a researcher in the CS department easier by sorting out administrative and computer problems. They do their jobs so well that potential stumbling blocks never seem to get in the way. In particular I would like to thank Sharon Burks, Catherine Copetas, Jim Moody, and Bill Ross.

Of course my years at CMU wouldn't have been enjoyable—perhaps not even tolerable, sometimes—without the great friends and officemates with whom to share troubles and joys. I was especially glad to have Keith Gremban around to run with, share beers with, and talk to about football and graduate student life. Dan Kuokka was also a good friend with whom to face Qualls and other graduate student trials. Finally, I need to thank everyone who gave me general support in this endeavor. Thanks to my parents, and thanks to my wife Carole for putting up with some long work hours and always beaming me positive thoughts.

Table of Contents

1. Introduction	1
1.1. Driving Levels	3
1.2. Why Perception is Hard in Complex, Dynamic Domains	6
1.3. The Cost of General Perception for Driving	7
1.4. Selective Perception for Driving	11
1.5. Thesis Outline	11
2. The PHAROS Traffic Simulator	13
2.1. Using a Simulated World	13
2.2. The Street Environment	14
2.3. PHAROS Driving Model	16
2.4. Simulator Features	17
2.5. Interface to a Simulated Robot	20
2.6. Summary	21
3. The Ulysses Driving Model	23
3.1. The Need for a Computational Model	23
3.2. Related Work	24
3.3. Tactical Driving Knowledge	26
3.3.1. A Two-Lane Highway	26
3.3.2. An Intersection Without Traffic	30
3.3.3. An Intersection with Traffic	35
3.3.4. A Multi-lane Intersection Approach	40
3.3.5. A Multi-lane Road with Traffic	43
3.3.6. Traffic on Multi-lane Intersection Approach	47
3.3.7. Closely spaced intersections	49
3.4. Interface to the World	49
3.4.1. Tactical Perception	49
3.4.2. Modeling Time	51
3.5. Model Limitations	51
3.5.1. Embedded Assumptions	52
3.5.2. Other Limitations	53
3.6. Summary	53
4. Building a Driving System	55
4.1. Integrating Driving Levels	55
4.1.1. Current Implementation	55
4.1.2. A Future System	58
4.2. Execution Control	59
4.3. Perception Control	60

4.4. Summary	62
5. Ulysses 1: Perceptual Routines	65
5.1. Defining the Perceptual Language	65
5.1.1. Routines	65
5.1.2. Related Work	66
5.1.3. How Routines Are Used	67
5.2. The Cost of Using Routines	73
5.2.1. General Cost	73
5.2.2. Driving Experiments	74
5.2.2.1. Left Side Road	75
5.2.2.2. Unordered Intersection	75
5.2.2.3. Four-lane Highway	78
5.2.2.4. Intersection With Traffic Lights	78
5.2.2.5. Multiple Intersections	78
5.3. Summary	81
6. Ulysses 2: Ignoring Redundant Constraints	83
6.1. Ulysses-2 Data Structures	85
6.1.1. The Corridor	85
6.1.2. The Inference Tree	87
6.2. The Search Algorithm	90
6.2.1. Tree Creation and Bounds Propagation	90
6.2.2. Tree Evaluation	95
6.3. Examples	99
6.4. Experimental Results	107
6.4.1. Left Side Road	107
6.4.2. Unordered Intersection	111
6.4.3. Four-lane Highway	111
6.4.4. Intersection With Traffic Lights	111
6.4.5. Multiple Intersections	121
6.5. Related Work	121
6.6. Summary	125
7. Ulysses 3: Modeling World Dynamics	127
7.1. Data Structure Modifications	127
7.2. Algorithm Modifications	128
7.3. Experimental Results	128
7.3.1. Left Side Road	130
7.3.2. Unordered Intersection	135
7.3.3. Four-lane Highway	135
7.3.4. Intersection With Traffic Lights	135
7.3.5. Multiple Intersections	145
7.4. Discussion	149
7.5. Summary	152
8. Conclusions	153
8.1. Summary	153
8.2. Contributions	154
8.3. Future Work	155
Appendix A. Computation Cost for the General Perception Model	159

Appendix B. The Cost of Perceptual Routines	167
Appendix C. Routine Costs in the Left Side Road Scenario	173
Appendix D. Bounds Propagation Algorithms	177
Appendix E. Input Selection Algorithms	187

Chapter 1

Introduction

One of the aims of mobile robot research is to produce robots capable of performing complex tasks in real-world environments. These robots require intelligent reasoning systems to assess the meaning of different objects around the robot and choose an action. In the past, robot planning systems have typically assumed that a perception module could refresh the entire world model whenever planning was needed, or whenever the world changed. However, in a complex, dynamic environment this assumption is completely unrealistic because perception is difficult and computationally expensive. Robots in these domains cannot use the traditional planner paradigm, but instead need a new system design that uses the reasoning component to control perceptual actions. In this thesis I describe how reasoning can be integrated with perception, how task knowledge can be used to select perceptual targets, and how this selection dramatically reduces the computational cost of perception. These concepts are illustrated for the domain of driving by a robot driving program called Ulysses.

The difficulty of making a complete world model from sensed data can easily be illustrated. Consider the driving scene shown in Figure 1-1. It contains vehicles in various locations and poses. To recognize an arbitrary vehicle, a perception system would have to consider

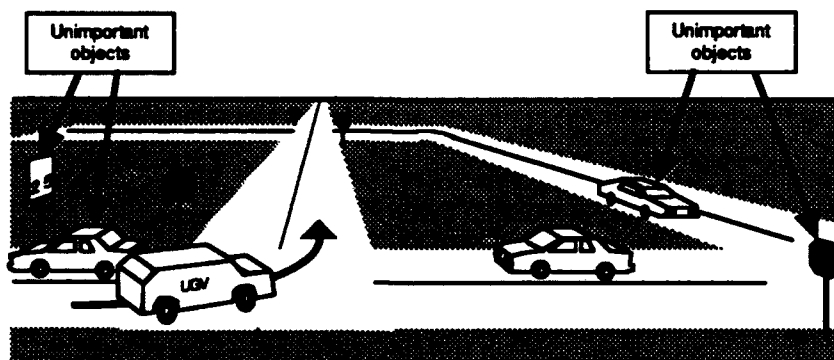


Figure 1-1: A driving scene with vehicles.

variations in vehicle shape, color, and illumination, as well as anomalies due to surface markings, reflections, transparent surfaces, occlusions, etc. The vehicles are different

distances and directions from the robot, and so appear in different locations in the robot's image of the scene. A perception system attempting to find *all* vehicles in the scene would have to search all possible locations, ranges, and poses, as we sketch in Figure 1-2. To interpret the scene completely, the perception system would also have to locate and identify all other traffic objects, with similar variations. The combinatorics of all of these factors together make such *exhaustive* perception far too expensive for a robot driver to do in a dynamic driving situation. Even humans cannot do this.

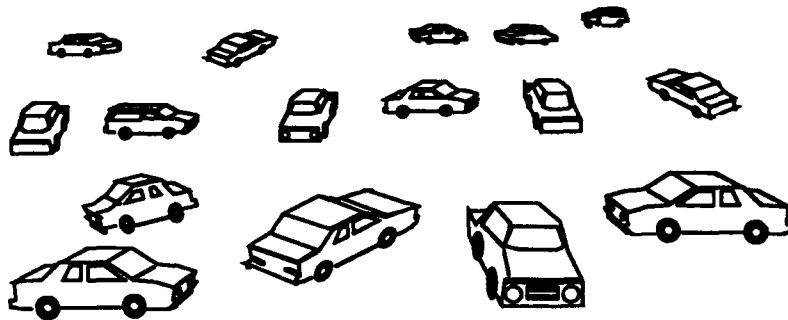


Figure 1-2: Where a naive perception system looks for cars.

Fortunately, it is not necessary for a robot driver to update its world model completely. Figure 1-1 illustrates that although traffic objects may be found in many locations throughout a scene, it is wasteful to look for all of them because they are not all important to the robot. In this thesis I demonstrate how domain knowledge can be encoded in a principled way to sharply reduce the number of objects for which the robot must look and constrain where it must search. Figure 1-3 illustrates this limited visual search. The driving system described in this thesis employs three mechanisms to control perceptual costs: First, perceptual routines are used to describe where to find the important traffic objects. Second, a new algorithm is used to search an inference tree to identify the minimum set of perceptual inputs needed to determine an action. Third, knowledge about domain dynamics is explicitly encoded in the inference tree so that perceptual inputs are refreshed only when necessary. The result is a limited visual search that is up to twelve orders of magnitude cheaper than exhaustive perception. This thesis describes the driving task and shows how each of these mechanisms can be applied to reduce the cost of perception.

In the remainder of this chapter I describe in greater detail the nature of the perception problem for driving. The next section discusses different levels of the driving task and describes the level addressed by this thesis. Section 1.2 explains why perception is significantly harder in a domain such as driving than it is for simpler robot tasks. Section 1.3 presents an estimate of the actual computational cost of general perception for driving, and illustrates why the general approach is effectively impossible.

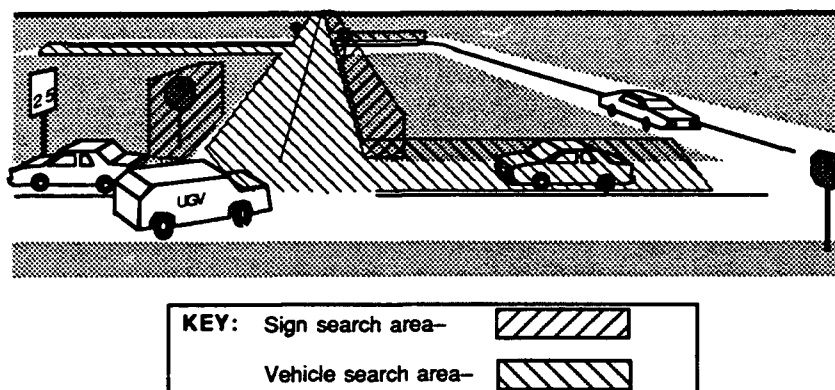


Figure 1-3: Selective visual search.

1.1. Driving Levels

The driving task has been characterized as having three levels: strategic, tactical and operational [Michon 85]. These levels are illustrated in Table 1-1. The highest level is the

Level	Characteristic	Example	Existing model
Strategic	Static; abstract	Planning a route; Estimating time for trip	Planning programs (Artificial Intelligence)
Tactical	Dynamic; physical	Determining Right of Way; Passing another car	Human driving models
Operational	Feedback control	Tracking a lane; Following a car	Robot control systems

Table 1-1: Characteristics of three levels of driving.

strategic level, at which a route is planned and behavioral goals are developed for the vehicle. These behavioral goals may be based on route selection and driving time calculations, for example. The strategic goals are achieved by activities at the middle, *tactical*, level, which involves choosing speed and steering maneuvers in the immediate situation. The maneuvers selected at the tactical level are carried out by the *operational* level of speed and steering control. This thesis addresses the tactical level.

Substantial progress has already been made in automating various parts of the driving

task, particularly at the operational and strategic levels. At the operational level, various research groups have been investigating vehicle steering and speed control for at least 30 years [Cardew 70, Fenton 91, Gardels 60, Lang 79, Masaki 92, Oshima 65, Shladover 91]. This work generally assumes that special guides are placed in the roads, so the vehicles are not completely autonomous. In the 1980's, robots began driving on roads autonomously [Dickmanns 86, Kluge 89, Kuan 88, Pomerleau 89, Texas Engineering Experiment Station 89, Thorpe 88, Tsugawa 79, Turk 87, Waxman 87]. The different robots have strengths in different areas; some are fast (100 kph), while others are very reliable in difficult lighting or terrain conditions. Autonomous vehicles can also follow other vehicles, with or without a special guide device on the lead vehicle [Bender 69, Cro 70, Gage 87, Kehtarnavaz ed, Kories 88, Texas Engineering Experiment Station 89].

These existing robot systems can perform the fast control functions needed at the operational level, but they are inadequate for the more complex reasoning needed at higher levels. Consider the driver approaching the intersection in Figure 1-4 from the bottom. A robot with current operational capabilities could track the lane to the intersection, find a path across the intersection, and then start following a new lane. The robot could also perhaps detect the car on the right and stop or swerve if a collision were imminent. However, in the situation illustrated, a driver is required to interpret the intersection configuration and signs and decide who is supposed to cross the intersection first. Current robot driving programs cannot perform this tactical task. Nevertheless, it is the success of autonomous vehicles at the operational level that motivates us to study the tactical level.

Computers have also been used to automate strategic-level driving functions. Map-based navigation systems have advanced from research projects to commercial products in the last decade [Belcher 89, Elliott 82, Kawashima 91, Rillings 91, Sugie 84, von Tomkewitsch 91]. Artificial Intelligence programs have been applied to many abstract problems, including the planning of driving errands [Hayes-Roth 85]. These planners work on static problems using information conveniently encoded in a database ahead of time. Let us consider again the driver of Figure 1-4. A strategic planner may have already determined that this driver is going to a pizza store, and that he must turn left at this intersection, and that he should drive quickly because he is hungry. However, the planner (probably) does not know ahead of time what traffic control devices (TCD's) are present at this intersection, and certainly cannot predict the arrival of cross traffic. In general, the world is uncertain and dynamic at this level, and the driver must continually use perception to assess the situation. Situations may change frequently enough that "planning" many future actions may in fact be pointless. Strategic planners are not designed to deal with these tactical problems.

Tactical driving requires features of both real-time systems and symbolic reasoning systems. Decisions must be made dynamically in response to quickly changing traffic

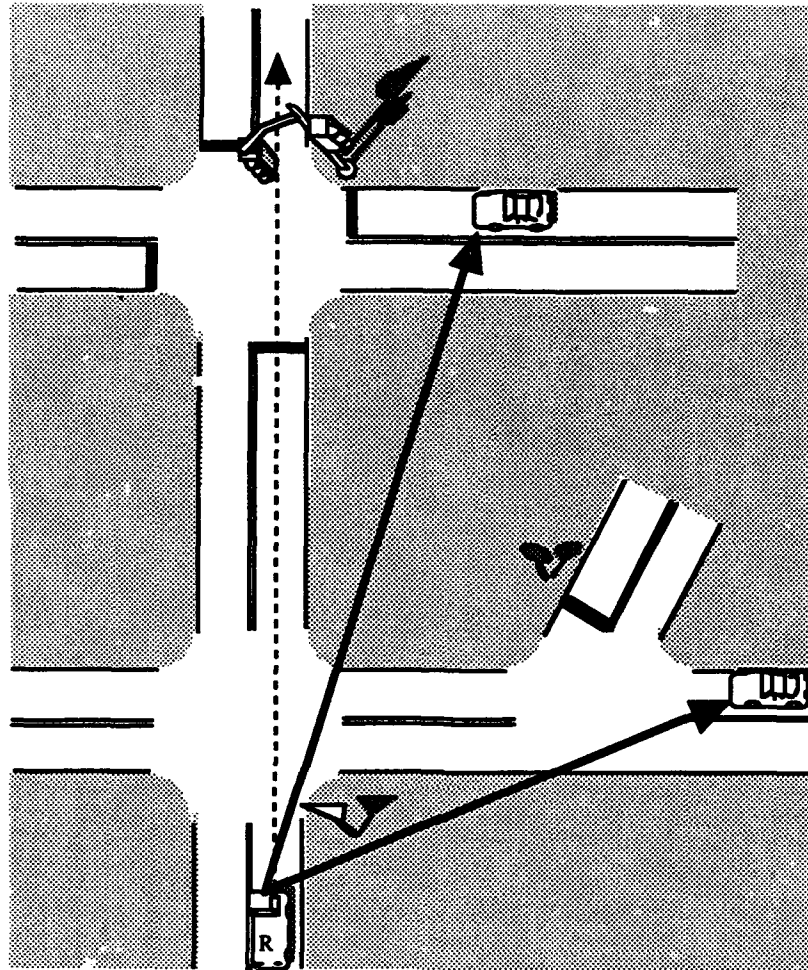


Figure 1-4: Example of tactical driving task: driver approaching crossroad.

situations. Since the situation is unknown ahead of time, the robot driver must get data by *looking* for signs and cars when it gets to the intersection. The robot must collect the appropriate data to interpret the situation and reason out what action to take. The reasoning process can be somewhat complicated, as illustrated by Figure 1-4. In this figure, the robot is required to yield by the nearby sign, but it must make a judgement about the distance, speed, and likely constraints on the vehicle to the right to decide whether it (the robot) should stop. In addition, if the robot does not stop, it must consider the signal at the next intersection and whether it must stop at that intersection. This decision may depend on the other car as well as the signal.

In order to study perception for tactical driving, it was necessary first to study the driving task itself in some detail. To this end, I have developed a microscopic traffic simulator called

PHAROS [Reece 88] (for Public Highway And ROad Simulator) to define the environment, and a computational model of driving called Ulysses [Reece 91]. Ulysses describes what tactical actions a robot driver should take, given the particular traffic objects around the robot. While Ulysses does not model every possible world state, it does capture the essence of car-following, lane selection, and right-of-way determination problems. Furthermore, it represents knowledge in a way that allows new driving rules to be added, and old rules to be changed to accomodate different assumptions. PHAROS and Ulysses are discussed in more detail in Chapters 2 and 3.

This thesis describes programs that implement Ulysses—that is, they implement just the tactical driving level. The strategic level was treated as an independent driving component that simply gives a complete route plan to the tactical level. The operational level was assumed to be a self-contained component that executes tactical maneuver commands. In an actual, complete driving system, this simple top-down pattern of communication would probably be inadequate, as it would be necessary for lower levels to occasionally share information directly with higher levels. Thus Ulysses would require some extensions to fully cover the tactical level in a complete driving system. I discuss how Ulysses would fit into a complete driving system in Chapter 4.

1.2. Why Perception is Hard in Complex, Dynamic Domains

Robots in some domains can use minimal vision systems. Instead of sensing, robots can sometimes make extensive use of an internal world model to predict and plan all their actions from an initial state. In fact, if the initial world state is pre-compiled, no perception is necessary at all. In other domains the environment is much more complex, but the task requires only simple perception. Monitoring scalar values, navigating with proximity sensors and tracking blobs with coarse imaging sensors are examples.

Other domains require perception of objects, but the objects and backgrounds are still simple. The environment doesn't change except for robot actions. Decades of computer vision research have shown that vision is very difficult even with these simplifying conditions. Recently several reports have estimated the complexity of vision computations in such domains. Template matching is polynomially complex in the size of the template (model) and image [Tsotsos 87]. Matching processed image features to model features is in general exponentially complex [Grimson 90], but in some cases polynomial in the number of scene and model features [Huttenlocher 90]. Interpreting all features in a scene together requires searching the space of all possible interpretations, which is exponentially large:

$$\text{Number of interpretations} = (\text{Number of region classes})^{\text{Number of regions}}$$

Still, in a static environment, it is not unreasonable to take a long time to look at the world.

Robot perception is even harder in complex, dynamic domains. This difficulty is due to several domain characteristics:

- The world state is not known in advance and cannot be predicted. Therefore the robot must observe the environment continuously.
- Objects play different roles. For example, they are not all just obstacles. The driving environment has many objects and types of objects—cars, roads, markings, signs, signals, etc.
- The appearance of objects changes due to many factors, including location in the field of view, range, size, orientation, shape, color, marking, occlusion, illumination, reflections, dirt, haze, etc. The appearance can vary as much as the product of all these factors.
- The environment is cluttered and distracting. It contains many background features that can be confused with the features of important objects.
- Domain dynamics place time constraints on perceptual computations.

The robot must search through a lot of data—from a rich sense such as vision—to discern objects in such a confounding environment. This is why perception is so computationally expensive. For example, previous work in autonomous road following and car and sign recognition [Akatsuka 87, Crisman 88, Ettinger 88, Griswold 89, Kluge 88, McKeown 88] has shown that even perceiving "simple" individual traffic objects in constrained situations is difficult. If the computational cost is many orders of magnitude too high for the time and resources available, then we can say that perception is *effectively intractable*.

Various techniques have been used to reduce the computation time for object recognition. These include using easy-to-find object features; using distinguished features that quickly discriminate between objects; using geometric or other constraints between features; using coarse input data (followed by fine data in small areas); and computing features in parallel. Without these techniques, even relatively simple vision tasks would be impractical. To date, these approaches have been demonstrated only for simple tasks in complex environments (for example, obstacle avoidance and road following), or for complex tasks in controlled environments (for example, bin picking). We believe that a powerful perception system that works as quickly and effectively as the human system must use all of these methods. However, general perception is still inadequate in complex, dynamic domains.

1.3. The Cost of General Perception for Driving

The previous section discussed why perception is expensive in general in a complex, dynamic domain. This section presents a specific analysis of perceptual costs for driving in traffic. In particular it estimates perceptual costs assuming that general, exhaustive perception is used—i.e., the perception system searches in all directions around the robot for all the traffic objects the planner might potentially need. This analysis will demonstrate the futility of an exhaustive approach to perception for a real problem, and provide a basis for evaluating the active vision techniques to be introduced later.

No one has actually built a traffic scene interpretation system, so we cannot determine the exact complexity of the problem. Two opposing factors make analysis of this unsolved problem especially difficult. First, it is possible that further research and experimentation will yield a very simple and inexpensive method of interpreting some aspect of the scene. On the other hand, real-world computer perception problems tend to be much more difficult than originally expected. For example, the NAVLAB project here at Carnegie Mellon investigated "basic" road following for several years. The simple road-recognition techniques that were attempted first often failed when conditions weren't ideal. For the purposes of this analysis, I have hypothesized a generic scene interpretation system that uses standard image processing and interpretation techniques. The remainder of this section discusses assumptions about the environment, task, sensors, and processing steps and the resulting cost calculation.

The environment. In order to study how hard it is to visually search for objects, we must first understand what there is to see. In my research, I did this by developing a model of the driving environment and implementing it in PHAROS. PHAROS contains detailed representations of roads, lanes, intersections, signs, signals, markings, and cars. Although PHAROS provides a rich setting for driving experiments, it also makes important abstractions that simplify and limit the driving problem for our work. For example, all roads are structured with lanes; also, there are no pedestrians or bicylists in PHAROS.

The analysis of perceptual cost also considers factors not explicitly represented in PHAROS. The environment is assumed to have shadows, trees, clouds, occluding objects, textures, and reflections. Although these factors are not explicitly modeled in PHAROS, they are present in the real world and therefore they are included in the model of perceptual cost. These factors prevent us from using cheap recognition algorithms based on single, uniform features. For example, although sign colors will be useful for segmenting out sign regions, there may be other objects that also have these colors. Additional information will be needed to distinguish the signs from the other regions in the image.

The task. Once we have created a representation of the world, we must specify what it means to drive. Although driving laws are published in books [Legislative Reference Bureau 87], and the driving task has been thoroughly analyzed in isolated situations [McKnight 70], there are no previously existing driving descriptions that actually specify what actions to take at any time. The Ulysses model of tactical driving does encode what action to take in any situation in the PHAROS world. Ulysses is a sophisticated model that incorporates knowledge of speed limits, car following, lane changing, traffic control devices, right of way rules, and simple vehicle dynamics. Ulysses is the definition of the driving task for this research.

A general perception system has to find all traffic objects of potential interest to the driving planner. For Ulysses, these objects include

- Road regions.
- Road markings, including, for example, lane lines and any other markings that would indicate turn lanes.
- Vehicles. Vehicles may be turned at various angles. Velocity estimates are required.
- Traffic signs. The robot must recognize signs that are facing up to 45 degrees away from the line of sight. Signs may be up to 7m above the roadway. Only a limited set of regulatory and warning signs are included in this task. The robot must sometimes recognize Stop and Yield signs from the back at intersections [Reece 91].
- Traffic signals. The robot must recognize signals that are facing up to 45 degrees away from the line of sight. Signals may be about 6m above the roadway.

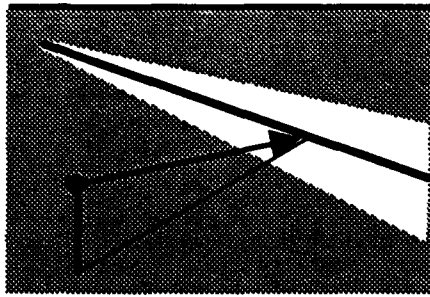
These objects are characterized in more detail in Appendix A.

Sensors. Since the environment is so varied in appearance, this analysis assumes that several types of sensors will be used to collect data. It assumes the robot has cameras and laser rangefinders whose images can be registered. This combination is complementary in that a rangefinder is almost immune to the illumination changes across an object which confuse color based segmentation. A camera, on the other hand, can discern markings and other important regions on uniform surfaces like signs [Hebert 88]. Since the robot must find objects in all directions, it is assumed to have sensors pointing in several directions. Sensor aiming time is thus not an issue addressed in this thesis.

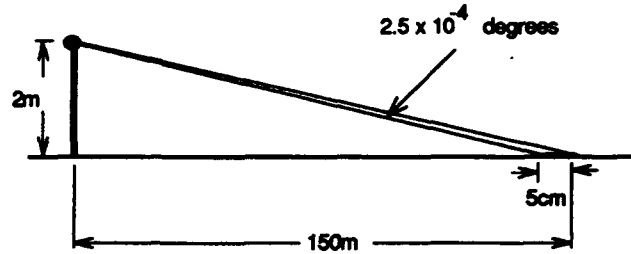
Since it is not practical to consider perceiving the entire world, I arbitrarily set a range limit on the sensor system. In some driving situations it may be desirable to see long distances ahead; for example, signals are supposed to be visible from 218m away on a road with 100kph traffic [Federal Highway Administration 78, pg. 4B-11], and the sight distance needed for passing is given as 305m at this speed [AASHTO 84, pg. 147]. While Ulysses is capable of driving on simulated highways, for this work I have concentrated on arterial urban streets where the visual environment is more diverse. Since streets have lower speeds than highways, I have chosen 150m as the perceptual range limit.

Given the arbitrary range limit of 150m, the sensors are assumed to have the resolution (and range) required to identify objects at full range. The resolution needed is determined by the smallest object. Since flat, horizontal objects like road markings are turned away from the line of sight and greatly foreshortened, they appear to be the smallest objects. Figure 1-5 shows that to cover a foreshortened 10cm-wide lane line with 2 pixels at 150m, a camera mounted 2m above the ground would have to have a resolution of 2.5×10^{-4} degrees per pixel. Appendix A discusses the resolution requirements of different traffic objects in more detail.

Data processing. Interpreting general traffic scenes will be very difficult. A general



a.



b.

Figure 1-5:

Minimum pixel size: a. Elevated sensor looking at transverse lane line. b. Sensor angular resolution required to resolve a 5cm patch at a range of 150m. Sensors are mounted on a 2m-high platform.

perception system would have to extract many features from the image, including regions, boundaries, lines, corners, etc. The complexity of the environment would in general prevent perception from using single, uniform features to uniquely distinguish objects. Extraction would use intensity, color, optical flow, range, and reflectance data. Features must be grown, characterized, and merged. Scene features would be matched against features of traffic object models to identify traffic objects. This matching would be done in two stages; for example, sign surfaces will first be located before the sign message is examined at higher resolution. These processing steps are explained in more detail in Appendix A.

Based on work at CMU on robot driving, and on an examination of the literature, I have estimated an approximate formula that relates image size to computational cost in this domain. As explained in Appendix A, the formula is

$$\text{Cost} \approx 1.1 \times 10^4 P + 1.1 P^2 + 1.7 \times 10^{-5} P^3 \text{ operations} \quad (1.1)$$

where P is the number of pixels in the image. The unit cost is an arithmetic operation on a data value or pixel. The linear term reflects the computations to find colors, edges, optical flow, etc. at each pixel; the squared term comes from pixel clustering and comparing pairs of features to each other; and the cubic term is our estimate of the cost of matching, using constraints to prune the search space. These computations are all detailed in Appendix A. Traffic objects are small enough to require about 1cm resolution, which translates to about 8×10^7 pixels. The net cost is then

$$\begin{aligned} \text{Cost} &\approx 8.9 \times 10^{11} + 7.2 \times 10^{15} + 9.0 \times 10^{18} \\ &\approx 9.0 \times 10^{18} \text{ operations.} \end{aligned}$$

As will be discussed later, Ulysses is assumed to process a new image every 100 milliseconds; so Ulysses would require about 10^{20} operations per second if it used naive, exhaustive perception. A "fast" computer that can perform a billion operations per second would thus be almost 11 orders of magnitude too slow to analyze the scene. Even if these estimates are off

by several orders of magnitude, it is clear that a general approach to perception is intractable.

1.4. Selective Perception for Driving

General perception is clearly far too slow for a driving robot. This thesis describes three ways in which perceptual requirements and costs can be reduced. These methods have been implemented in three programs, Ulysses-1, -2, and -3.

1. **Ulysses-1** does not depend on a world model but instead explicitly requests all information from the perception component. Since the Ulysses driving model bases decisions on objects that have specific spatial relations to other objects (e.g., the car *ahead in the lane*), perceptual actions are *routines* that use reference objects (e.g., a lane) to search appropriate areas of the scene. Thus perceptual requests can return important objects directly and eliminate the need for an additional geometric search (e.g., which of the cars found is in this lane?).
2. **Ulysses-2** also uses perceptual routines, but also tries to find the minimum set of requests necessary to determine the correct action. The Ulysses driving model is represented explicitly as an inference tree that relates uncertainty about traffic objects to uncertainty about actions. Ulysses-2 searches the inference tree to find the most critical leaf in the tree, and calls the appropriate perceptual routine. The effects of the new information are propagated up the tree. This selection and sensing process is repeated until there is no uncertainty about what action should be taken.
3. **Ulysses-3** uses the same mechanism used in Ulysses-2 but also maintains a world model. Facts in the model are time-stamped, and Ulysses-3 uses domain knowledge to reason about how object characteristics change over time. The inference tree search process automatically determines when changing objects must be sensed again to reduce uncertainty.

Measurements of perceptual requests in simulated driving situations show that these techniques reduce the cost of perception between eight and twelve orders of magnitude from general perception.

1.5. Thesis Outline

This research covers new ground in both robot driving and selective perception. In the following chapters, the thesis discusses both of these areas. Chapter 2 provides more detail about the PHAROS simulator, which was used not only to define the driving environment but also to study driving behavior, test perceptual interfaces, and test the implemented Ulysses driving programs. Chapter 3 describes the Ulysses driving model. Chapter 4 discusses several robot systems issues concerning the implementation of a driving program: how driving levels are integrated, how reasoning and perception operate in real time, and how selective perception is incorporated into the architecture. The next three chapters describe the three techniques I propose to control the cost of perception. Ulysses-1, which uses perceptual routines, is presented in Chapter 5. Chapter 6 describes Ulysses-2, which

reduces sensed-data requirements by purposefully applying driving knowledge. Chapter 7 shows how Ulysses-3 uses knowledge of world dynamics to further optimize search and reduce sensing needs. Finally, the thesis concludes with a discussion of the implications of these selective perception techniques and some future extensions to them.

Chapter 2

The PHAROS Traffic Simulator

The first step in this robot driving research was the development of a detailed, microscopic traffic simulator called PHAROS. It is "microscopic" because the actions of each vehicle are modeled separately. This chapter discusses the purposes of the simulator, the world model, the operating characteristics, and the relation to the robot driving program.

2.1. Using a Simulated World

PHAROS serves several purposes. First, it models the physical, observable environment. Such a model is a necessary step in modeling the overall driving task. The environment model specifies what objects are important and defines the lowest level abstractions the robot can use in reasoning about the world. PHAROS's representation of physical objects, together with its simulation of perception, determine what information a robot can get from the (simulated) world. Second, PHAROS provides a testbed for developing driving rules. The vehicles in PHAROS do not use simulated perception, but can directly use the information in the simulator's data structures. Thus driving logic is separated from the issue of perception and can be developed in an idealized context. The performance of a driving model can be evaluated by observing the vehicles moving in the simulated environment. Finally, PHAROS is the test environment for the Ulysses driving program implementations. For the robot, PHAROS accepts perceptual requests, simulates the perception and returns data. PHAROS also accepts action commands and simulates robot motion in the world.

There are several advantages to driving in simulation before trying to drive an actual vehicle. First, a simulator is flexible. It is possible to generate almost any traffic situation to test various kinds of driving knowledge. These situations can be recreated at will to observe how new driving knowledge changes a vehicle's behavior. Simulators are also convenient, because simulated driving is not dependent on working vehicle hardware, convenient test sites, or weather. Finally, simulated driving is safe and avoids the problem of placing other drivers at risk while testing the robot.

Traffic engineers have used simulators for a number of years in order to model the effects

of road and car design on traffic flow. I considered using some of these simulators for this research. However, most of the simulators in use today—for example, TRANSYT, PASSER, and SIGOP [Gibson 81]—are *macroscopic*, so do not model the behavior of individual vehicles. There are also microscopic simulators available, including SIMRO [Chin 85], TEXAS [Gibson 81], and NETSIM [Federal Highway Administration 80, Wong 90]. NETSIM in fact was the direct inspiration for PHAROS. However, NETSIM lacked several important features which were necessary for this research, including the following: detailed representation of spatial information; explicit representation of traffic control devices (TCD's) such as lines, road markings, signs, and signal heads; continuous vehicle movement in all situations including queue discharge, intersection traversal, and lane changing; and an animated graphical output. PHAROS does fill these requirements.

2.2. The Street Environment

PHAROS is a model of the street environment. All models of the world use abstractions so they can capture just the important characteristics of a problem. Model designers must find a compromise between model accuracy and excess complexity. We have attempted to encode many characteristics of the street domain into PHAROS, including enough geometric information to study the perceptual requirements of driving. While PHAROS includes many traffic objects, it uses strong abstraction and structure to simplify their description and use.

PHAROS represents traffic objects with abstract symbols augmented by a few parameters to provide important details. Figure 2-1 illustrates how some objects are encoded. A curved road segment is described by a cubic spline (8 parameters) and a road width. A sign is represented by a type (one of 8), an identifying number, and a position along the road. The general shape and color of the sign is determined by its general type; the lateral distance from the road edge is assumed to vary only a little, so is not encoded at all. PHAROS describes a signal head by its location at the intersection (one of 5 generic positions), whether it is vertical or horizontal, the number of lenses, the symbols on the lenses, and the color of the lenses.

PHAROS groups traffic object symbols into structures to give them meaning for driving. It would be difficult to determine how streets connected or whether a car could move to an adjacent lane if the lane and line objects in the database were not organized. PHAROS connects objects to one another to form hierarchies and networks of related objects. Figure 2-2 shows how different street objects are connected to one another.

The abstraction and structure used in PHAROS limit the accuracy with which it can simulate the world. For example, PHAROS cannot easily simulate roads without marked lanes. PHAROS vehicles cannot abandon the lane abstraction to drive around a broken










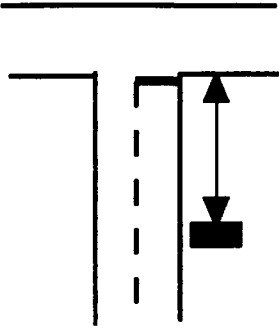
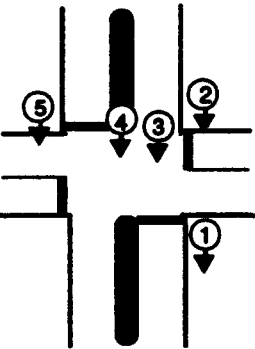






Symbol	Example Parameters				
Road Segment	Width	Curved Straight	Cubic spline paramters		
					
Sign	Identification #	Type	Position		
		 STOP  YIELD  Regulatory  Warning  Construction  Guide  Service  Recreation			
Signal Head	Location	Orientation	# Lenses	Lens symbol	Lens color
		 		    (or combination)	Red Yellow Green

Figure 2-1: Examples of traffic object encodings.

down vehicle at an intersection. However, our representation scheme does allow PHAROS to simulate many common traffic situations on highways and arterial streets. We also feel that the scheme could be extended to deal with situations that are beyond the current abstraction. Similarly, there are several features that could be added to PHAROS' world, such as pedestrians, cyclists, blinking signals, road grades, buildings, etc. Future versions of PHAROS may include such features.

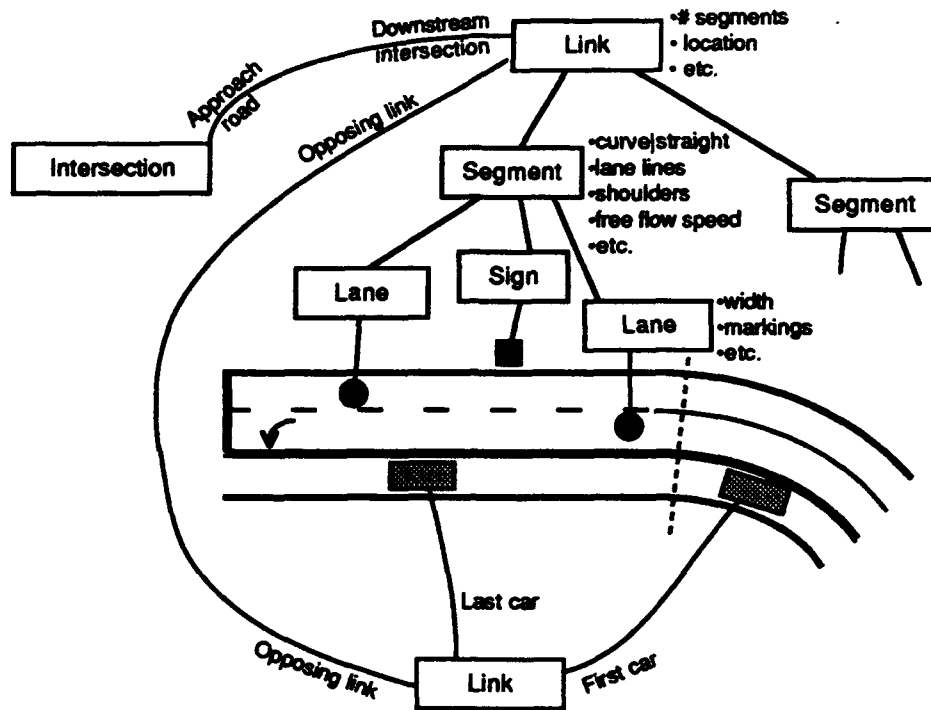


Figure 2-2: Structure of street objects.

2.3. PHAROS Driving Model

The driving decisions made by the PHAROS vehicles (which are called "zombies") are based on the logic in the Ulysses driving model, which is described in the next chapter. However, there are several differences between Ulysses and the PHAROS-controlled zombies. The key differences are in perception, interpretation, and reaction delays.

Perception. Zombies are not required to use simulated perception to get information from the simulated world. When PHAROS is running the decision procedure for the zombies, it uses the data structures directly. For example, to find a vehicle across the intersection from a zombie, PHAROS traces symbolic links from the zombie to its road, to the connecting road ahead, and to the rearmost car on that road. Tracing these pointers in the data structures is similar in concept to running perceptual routines, but much simpler in practice. In addition, zombies can inspect one another's data structures and establish crude interactions for merging, resolving right of way deadlocks, etc.

Interpretation. Several traffic situations that a driver encounters are difficult to interpret. Some situations, such as determining the traffic signal for other approaches or the intended turn maneuver of another car, are problematical for a driver because they cannot be observed directly. This type of problem is simple for zombies, because they have access to the entire database and are not limited by what they can see.

Other situations, such as deciding which signal head applies to the driver's lane, would normally require several observations followed by a reasoned guess. PHAROS shortcuts these difficult interpretation problems by encoding the semantics of situations directly into the database. For example, the traffic control for each lane is provided as part of the input data for the simulation and is stored with the other characteristics of the street. Lane channelization is also provided externally and stored.

Some situations are complex enough that the driving model does not have the expertise to recognize them. For example, it is difficult to reliably determine whether a blocking car in front of the robot is really broken down or merely part of a queue that extends to the intersection. Zombies determine if the car in front of them is in a queue simply by checking that car's "Queue" status; a zombie sets its own status to "enqueued" if it is close to the car in front and the car in front is already in a queue. The direct encoding of interpreted information, as well as the availability of physically unobservable state information, allows PHAROS to move zombies realistically without the complete expertise of a human driver.

Reaction Delays. PHAROS drivers are allowed perfect perception and access to special information so that they can choose actions as realistically—i.e., with human competence—as possible. However, zombies behave unrealistically if they are allowed to control speed too well. Therefore, PHAROS incorporates a reaction delay into zombie speed control. When zombies wish to decelerate quickly, they must switch to a braking "pedal" (state), and when they wish to accelerate again they must switch to an accelerator "pedal." There is a delay of 0.8 s, on average, in switching between pedals. This imposed reaction delay, when combined with the car-following law already discussed, results in fairly realistic behavior during car following, free-flow to car following transitions, free-flow to enqueued transitions, and queue discharge.

2.4. Simulator Features

Input. PHAROS generates a street database from a data file that it reads in at the start of each simulator run. Since PHAROS can represent a lot of detail in a variety of traffic objects, the data file can be fairly complex. The PHAROS user must encode the desired street environment into symbols and numbers and type them into the file. Figure 2-3 shows an example of how one face of a traffic signal head is encoded. The terms "face" and "lens"

```

face vertical 5 L9 LEFT MARGIN 0 0
  lens RED_SIGNAL SC_SOLID 8      1 1 1 1 1 0 0 1
  lens AMBER_SIGNAL SC_SOLID 8     0 0 0 0 0 0 1 0
  lens GREEN_SIGNAL SC_SOLID 8     0 0 0 0 0 1 0 0
  lens AMBER_SIGNAL SC_LEFT_TURN 8 0 0 0 0 1 0 0 0
  lens GREEN_SIGNAL SC_LEFT_TURN 8 0 0 0 1 0 0 0 0

```

Figure 2-3: Example of signal head encoding in PHAROS data file.

are keywords. The first line indicates that there are 5 lenses on this face of the signal head, that they are arranged vertically, and that the head is located on the left shoulder of the street named "L9." If the head were directly over a lane, the last two numbers on the first line would indicate which lane and where along the lane the head was located. The remaining lines describe each lens. The information includes the color, symbol, and diameter of the lens, and whether the lens is lit in each phase.

Encoding and typing this information is tedious and error-prone for multi-intersection networks. A future version of PHAROS should include a graphical input interface that generates the file automatically from pictorial descriptions.

Random traffic generation. Traffic in PHAROS is generated at designated "source" intersections at the edges of the network. The input data file specifies the average time between the arrival of platoons of cars and the average number of cars in a platoon. Provisions also exist for specifying a mix of vehicle types (passenger cars, buses, trucks, etc.), but PHAROS currently only simulates passenger cars. When individual cars are generated at run time, PHAROS assigns them a random "aggressiveness" parameter that modifies the default reaction time, desired free flow speed, and gap acceptance threshold. The parameters can also be used to break RoW deadlocks and force zombies in heavy traffic streams to allow other zombies to merge in. A newly created zombie's route through the network is determined by user-specified maneuver statistics at each intersection. I opted to use this technique rather than origin-destination statistics because our primary interest is in tactical driver behavior rather than network traffic flow. Thus we used the easiest method of specifying the exact traffic flow characteristics at each intersection.

Time. PHAROS has features of both a discrete time and a discrete event simulator. Some events, such as zombie creation and traffic signal changing, happen at irregular intervals; others, such as updating the positions of vehicles, occur at regular intervals. PHAROS maintains an event queue that keeps track of both types of events. The position-update events compute the vehicles' new positions exactly from their (constant) accelerations over the preceding time interval.

PHAROS vehicles are moved at regular intervals. Immediately after all vehicles are moved, each vehicle chooses a new acceleration and lane command for the next interval. This interval is an adjustable parameter, but remains fixed during a simulation run. For this research, the interval was always set at 100 milliseconds.

Display and user interaction. The output of PHAROS is an overhead view of the street environment with an animated display of the vehicles. This graphical output provides us with our primary means of evaluating the behavior of both zombies and robots. Figure 2-4 is a picture of the computer screen while PHAROS is running. The figure shows a black-and-

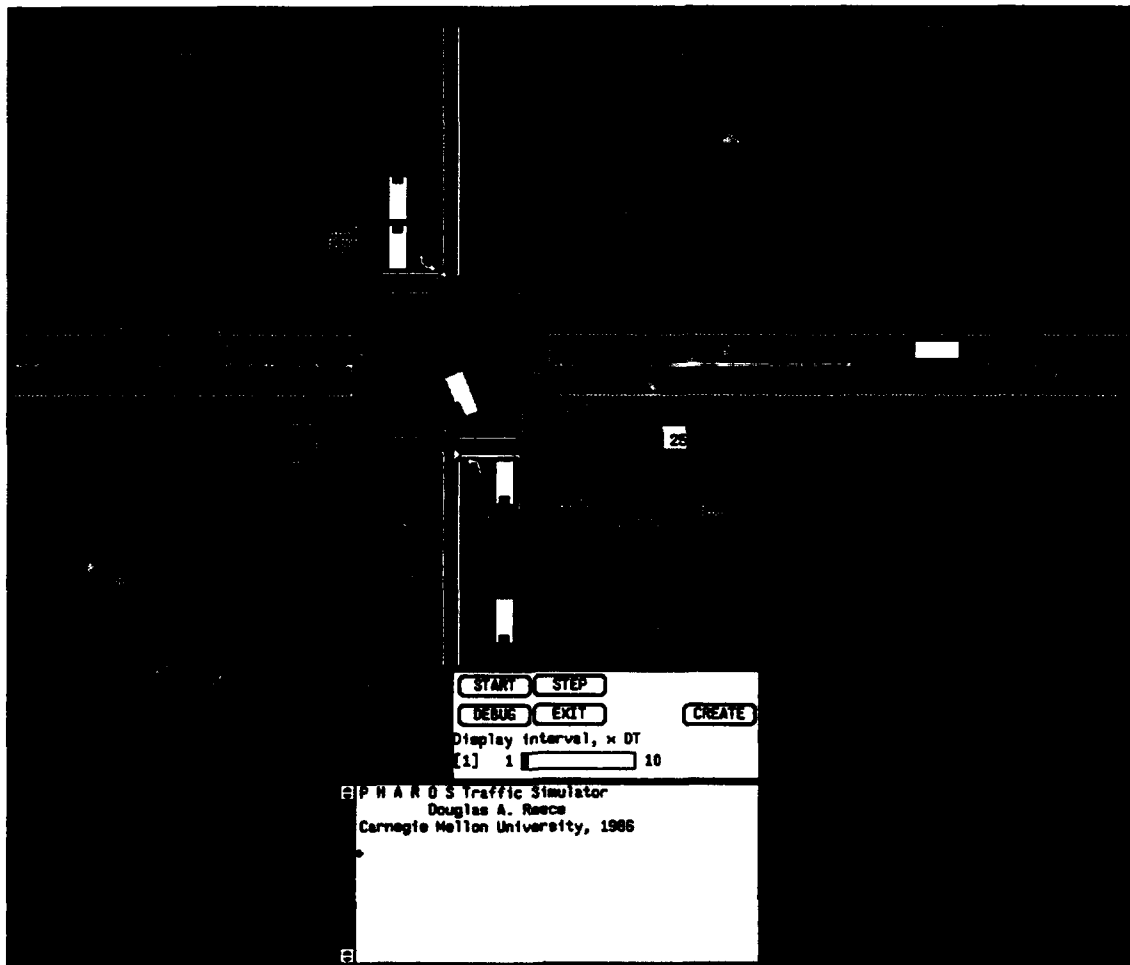


Figure 2-4: The output display of PHAROS.

white approximation of the computer's high-quality color display. In the lower-left corner is a window showing the entire street network schematically. A portion of the network (enclosed by a small rectangle in the schematic) is drawn at larger scale in the top-center of the screen. Cars are visible as white rectangles, with dark regions at the rear showing brake

lights and directional signals. The small window to the right of the network schematic shows simulator time. A user can interactively pan and zoom around the streets to examine various areas in more detail. It is also possible to select individual zombies and trace their decision processes. The simulator can be paused or stepped one decision cycle (100 ms) at a time.

To simulate a robot, a user selects the "button" on the screen marked "create;" PHAROS then pauses and asks where the robot should be injected into the street network. The robot is rendered in a unique color. We have created various displays to study the activity of Ulysses, including the location of each mark created by the perceptual routines, a count of the number of perceptual requests, and a chart of perceptual activity in each direction.

Performance. The simulator is written in C and runs on a Sun workstation under Sunview. A Sun 4/40 can drive a few dozen zombies in a one-intersection network about 3 times faster than real time¹. A network with 100 intersections and several hundred vehicles has also been created; this network runs at about half real-time speed.

2.5. Interface to a Simulated Robot

Later chapters will describe the robot driving program implementations. Although the logic in the robot driving program is similar to that for PHAROS zombies, the robot driving program is completely separate and could be entirely different. Figure 2-5 shows how PHAROS and a robot driving program are used together. PHAROS maintains a database describing the street environment and records for each car. PHAROS also controls the behavior of the zombies. The simulator executes a decision procedure for each zombie to determine its actions, and then moves it along the street. A robot is simulated by replacing the decision procedure with an interface to Ulysses. Ulysses runs as a separate program—sometimes on a different computer—and gets information by sending perceptual requests to PHAROS. The interface simulates perceptual functions and extracts data from the database. After Ulysses has the information it needs, it sends commands through the interface to the driving program. PHAROS then moves its representation of the robot using the normal movement routine. The requests for (simulated) perception and the commands for (simulated) control are the *only* communications between Ulysses and PHAROS. This arrangement ensures that Ulysses cannot "cheat" by examining the internal state of the simulator.

¹This speed is for the PHAROS alone; with a robot driving program also running, performance decreases considerably.

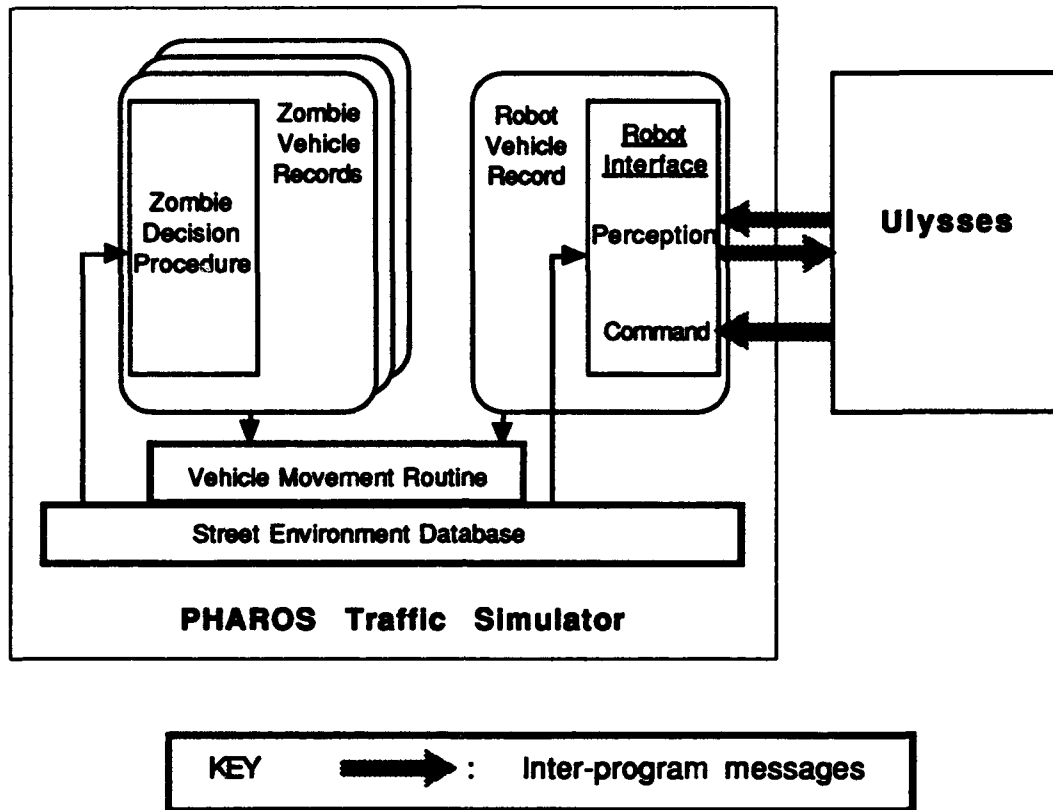


Figure 2-5: Ulysses control of a vehicle in PHAROS.

2.6. Summary

PHAROS is a microscopic model of the street environment. It defines the environment and serves as a testbed for robot driving experiments. The physical characteristics of streets, signs and other traffic objects are represented in detail. PHAROS also contains vehicles that move realistically through the network of streets. PHAROS can generate a variety of street networks from data files, and display cars moving through them with animated graphics.

PHAROS supports robot driving programs by simulating perceptual functions and implementing commands. The driving program takes over control of one vehicle in the simulator. PHAROS can simulate perception because it contains information about many important traffic objects; this capability is a significant extension to other existing

simulators, which do not model the visual environment in detail. The following chapters describe the driving programs used with PHAROS.

Chapter 3

The Ulysses Driving Model

This chapter describes the driving model I developed to define the driving task. After the introduction below, the chapter discusses related work in modelling drivers. This work comes mostly from the traffic engineering and psychology communities. The remainder of the chapter describes the model in detail.

3.1. The Need for a Computational Model

In order to study driving and implement an autonomous driver, we must have a model of the driving task. To be most useful, a driving model must be *detailed* and *complete*. A *detailed* model must state specifically what decisions must be made, what information is needed, and how it will be used. Such models are called computational because they tell exactly what computations the driving system must carry out. A *complete* driving model must address all aspects of driving. Traffic engineers and psychologists have long studied the driving task, including tactical driving, and have developed many good theories and insights about how people drive. Unfortunately, these models of human driving do not explain exactly what and how information is processed—what features of the world must be observed, what driving knowledge is needed and how it should be encoded, and how knowledge is applied to produce actions. A model must answer these questions before it can be used to compute what actions a robot should take.

Ulysses is a computational model of tactical driving. The program encodes knowledge of speed limits, headways, turn restrictions, and traffic control devices (TCD's) as constraints on acceleration and lane choice. The constraints are derived from a general desire to avoid collisions and a strategic driving goal of obeying traffic laws. Ulysses evaluates the current traffic situation by looking for important traffic objects. The observations, combined with the driving knowledge and a strategic route plan, determine what accelerations and lane changes are permitted in this situation. Ulysses also encodes preferences for actions. Preferences are also triggered by conditions observed in the scene. The program uses prioritized preferences to select an action from those allowed by the constraints. (This constraint- and preference-based knowledge representation scheme was inspired by similar schemes in other systems such as Soar [Laird 87] and Prodigy [Carbonell 91].) While this

model has limitations, it drives competently in many situations. Since it is a computational model, Ulysses shows exactly what information a driver needs at each moment as driving decisions are made. Furthermore, the program could in principle be tested objectively on a real vehicle.

3.2. Related Work

Psychologists, traffic engineers and automotive engineers have studied human driving a great deal. Their goal is to make cars and roads safer and more efficient for people. The result of this work is a multitude of driving models spanning all levels of driving. Michon identified seven types of models [Michon 85]: task analysis, information flow control, motivational, cognitive process, control, trait, and mechanistic. Each has different characteristics and addressed different aspects of driving.

Task analysis models. A task analysis model lists all of the tasks and subtasks involved in driving. The paragon of these models is McKnight and Adam's analysis [McKnight 70]. Their work provides an exhaustive breakdown of all activities on the tactical and operational levels. We have found this listing very useful for determining whether our model performs all of the necessary subtasks in various situations. However, a list of tasks alone is not sufficient for describing a driving model. This is because a task list does not address the dynamic relations between tasks. The list does not specify how the driver chooses a task in a given situation, or whether one task can interrupt another, or how two tasks might be performed simultaneously (especially if they require conflicting actions). The McKnight work also leaves situation interpretation vague. For example, two tasks require a driver to "observe pedestrians and playing children" and "ignore activity on the sidewalk that has no impact on driving." However, there are no computational details about how to discriminate between these situations.

There have been advances in task models recently in an effort to create intelligent driver aids [Malec 91, NadjmTehrani 91, Sandewall 90, Traffic Research Center 90]. These driver aids are part of the DRIVE and PROMETHEUS projects in Europe. This work covers many situations and subtasks, as McKnight and Adams' did. It describes a computational task model, and addresses the recognition of specific condition from observable traffic objects. However, these systems still seem to have the weakness that they treat each driving subtask as an independent, sequential procedure.

Information flow control models. Information flow control models are computer simulations of driving behavior. Early computer models of drivers were essentially implementations of task analysis models [Michon 85]. As such, they have the same weaknesses as the task analysis models. Microscopic traffic simulators such as

SIMRO [Chin 85], TEXAS [Gibson 81], and NETSIM [Federal Highway Administration 80, Wong 90] do not have these weaknesses because they perform multiple tasks simultaneously. They also have the ability to start and stop tasks at any time in response to traffic. For example, NETSIM evaluates the traffic situation frequently (every second) and makes a fresh selection of appropriate subtasks. NETSIM is not intended to be an accurate description of human cognitive processing, but it produces reasonable driver behavior in many situations.

NETSIM cannot be used as a complete driver model because it lacks several necessary components. There are gaps in its description of driver states; for example, cars change lanes instantly without having to drive across lane lines. Neither does NETSIM contain knowledge of how to interpret traffic conditions from *observable* objects. We also found unmodified NETSIM inadequate as a simulator testbed for robotics research because it cannot represent physical information such as road geometry, accurate vehicle location, or the location and appearance of TCD's.

Motivational models. Motivational models are theories of human cognitive activity during driving. Van der Molen and Botticher recently reviewed several of these models [van der Molen 87]. The models generally describe mental states such as "intentions," "expectancy," "perceived risk," "target level of risk," "need to hurry" or "distractions." These states are combined with perceptions in various ways to produce actions. Since motivational models attempt to describe the general thought processes required for driving, one would hope that they would form a basis for a vehicle driving program. However, the models do not concretely show how to represent driving knowledge, how to perceive traffic situations, or how to process information to obtain actions. Van der Molen and Botticher attempted to compare the operations of various models objectively on the same task [Rothengatter 88, van der Molen 87], but the models could be implemented only in the minds of the model designers. Some researchers are addressing this problem by describing *cognitive process models* of driving (for example, Aasman [Aasman 88]). These models are specified in an appropriate symbolic programming language such as Soar [Laird 87].

The remaining types of models have limited relevance to tactical driving. **Control models** attempt to describe the driver and vehicle as a feedback control system (e.g., [Reid 80]). These models are mainly useful for lane keeping and other operational tasks. **Trait models** show correlations between driver characteristics and driving actions. For example, drivers with faster reaction times may have a lower accident rate. This correlation does not describe the mechanism by which the two factors are related. Finally, **mechanistic models** describe the behavior of traffic as an aggregate whole. These models express the movement of traffic on a road mathematically as a flow of fluid [Drew 68]. This type of model cannot be used to specify the actions of individual drivers.

3.3. Tactical Driving Knowledge

3.3.1. A Two-Lane Highway

Figure 3-1 depicts a simple highway driving situation. In this scenario no lane-changing actions are necessary, but there are several *constraints* on speed.

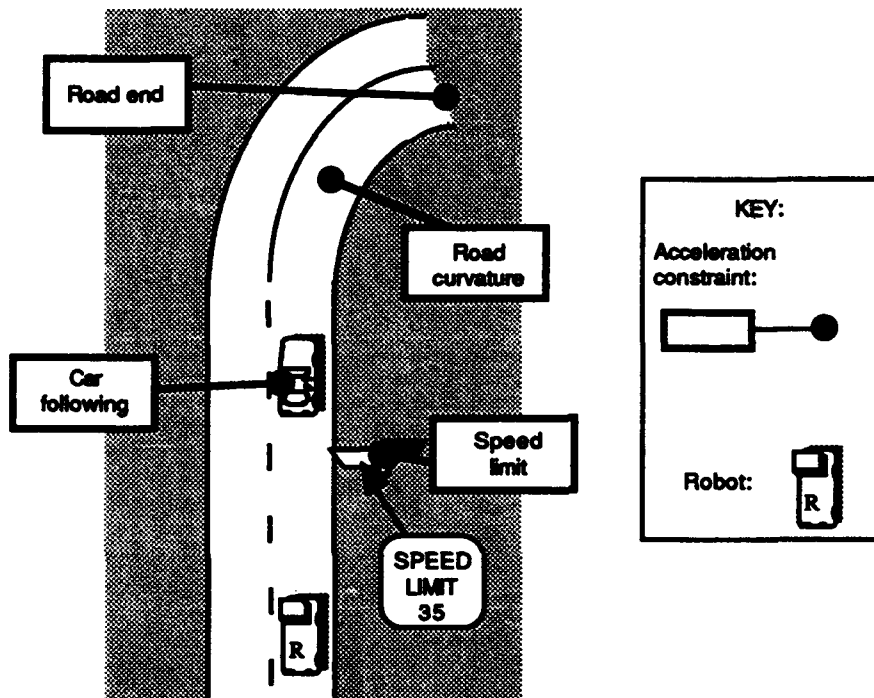


Figure 3-1: The two-lane highway scenario.

The first two constraints are derived from general safety goals—i.e., self-preservation. The speed of the vehicle must be low enough to allow it to come to a stop before the end of the road is reached; the speed on a curve must keep the vehicle's lateral acceleration below the limit imposed by friction between the tires and road surface. Although it is possible that these constraints can be met by the operational level systems of the robot, the prediction of future speed constraints from observations of distant conditions is in general a tactical activity. This is especially true if future conditions are detected not by tracking road features but by reading warning signs. Ulysses generates the road end and road curvature constraints by examining the corridor. The robot must be stopped at the end of the road if the road ends; furthermore, at each point of curvature change, the robot's speed must be less than that allowed by the vehicle's lateral acceleration limit. Ulysses also creates speed constraints at signs along the right side of the corridor that warn of road changes.

These constraints—a maximum speed at a point somewhere ahead in the corridor—are typical of the motion constraints generated by various driving rules. Given the robot's current speed, we could compute a constant acceleration value that would yield the desired speed at the desired point. However, it is also possible to satisfy the constraint by driving faster for a while and then braking hard. Figure 3-2 shows these two possibilities as curves X and Y on a graph of speed versus distance. The figure shows that any speed profile is

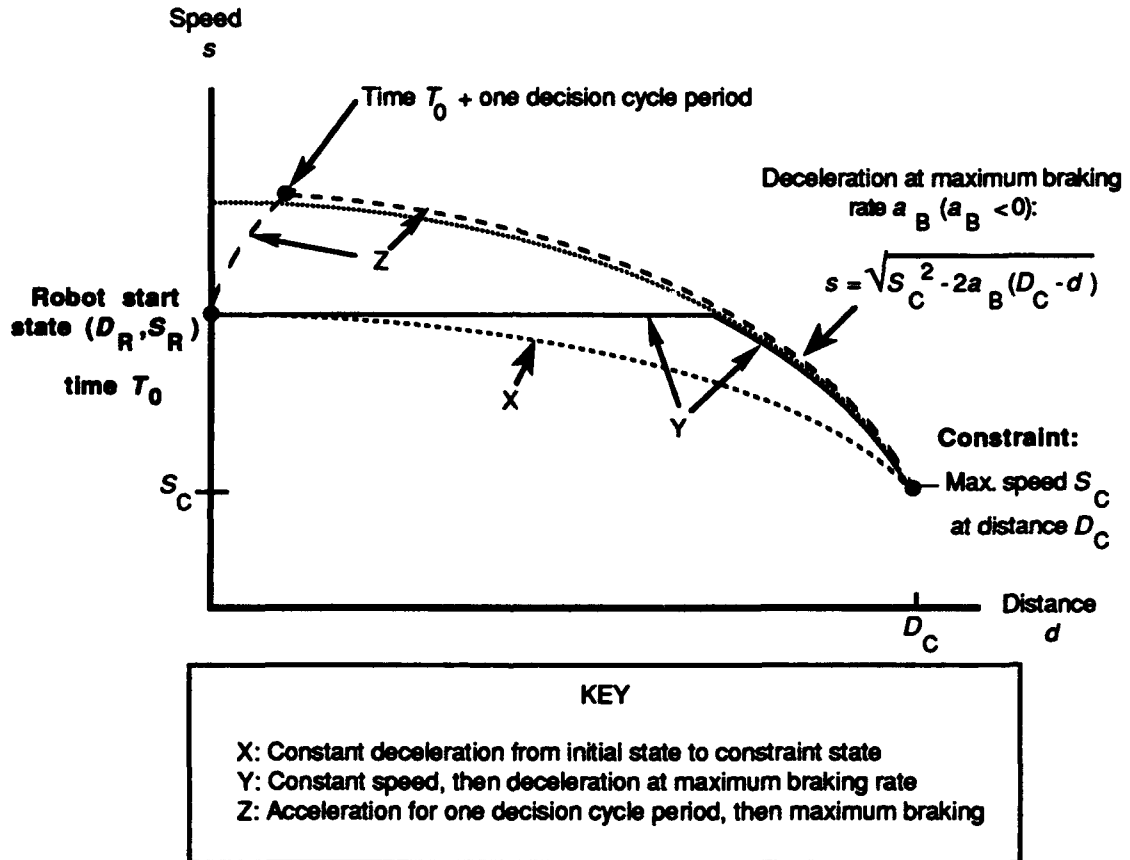


Figure 3-2: Deceleration options plotted on speed-distance graph

possible as long as it stays below the maximum braking curve. The profile that maximizes vehicle speed would rise instantly to the maximum braking curve and then follow the curve to the constraint point. However, the program cannot change the acceleration command at arbitrary times; it looks at situations and changes acceleration only at discrete intervals. Therefore Ulysses computes an acceleration that will cause the robot's speed to meet the maximum deceleration curve exactly at the next decision time. This is curve Z in the figure.

While the high-acceleration, high-deceleration policy shown by curve Z maximizes vehicle

speed, it seems to have the potential to cause jerky motion. In fact, the motion is usually smooth because the constraints relax as the robot moves forward and because actual vehicle acceleration is physically limited. For example, suppose that the robot's sensors could only detect the road 150 feet ahead. Ulysses would constrain the robot to a speed of 0 at a distance of 150 ft. However, when the robot moved forward, more road could be detected, so the constraint point would move ahead. Figure 3-3 shows how this "rolling horizon" affects speed. The robot starts at distance 0 with a speed of 45 fps. A decision interval of 1.0

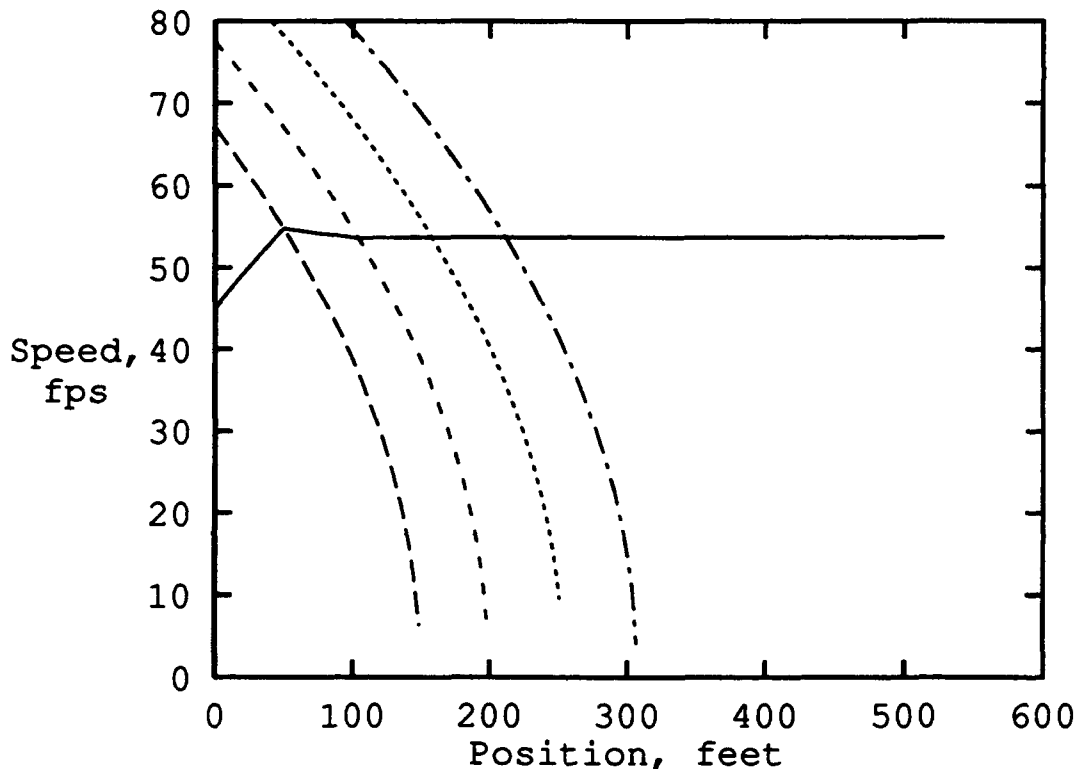


Figure 3-3: Robot speed profile (solid line) as it adjusts to a rolling road horizon. Maximum deceleration is -15.0 fps^2 , decision time interval 1.0 sec, horizon at 150 ft. The four dashed lines show, for the first four decision times, the maximum-deceleration profile to the current horizon.

seconds is assumed in order to show speed changes more clearly. The dashed lines show deceleration constraint curves (-15 fps^2) at the first four decision times. The solid line shows the robot's speed. After a small overshoot, the speed reaches an equilibrium and remains constant.

The basic acceleration constraint mechanism described in Figure 3-2 is modified slightly to achieve more realistic behavior over a range of conditions:

- A lower acceleration value—about half of maximum braking—is used to compute the constraint. The reduced value allows a gentler stop when the robot actually has to decelerate, and provides some margin for error.

- For car following (discussed below), the lead car's deceleration rate is assumed to be maximum, even though the robot uses less than maximum.
- If the constraint is instantaneous instead of applying at a distance—for example, the *current speed limit*—then acceleration is computed from $accel = \frac{constraint\ speed - robot\ speed}{dT}$, where dT is one decision cycle time.
- If the robot is so close to the constraint point that it will drive past the point during the next decision cycle, then the above equation is used rather than the method of Figure 3-2.

The next constraint illustrated by Figure 3-1 is that generated by legal speed limits. Ulysses maintains a state variable which records the current speed limit. The vehicle is allowed to accelerate so that it reaches the speed limit in one decision cycle period. Furthermore, Ulysses checks for "Speed Limit" signs along the right side of the corridor. Whenever the system detects a speed limit change, it creates an acceleration constraint to bring the robot's speed to the limit speed at the sign (as described above). As the robot nears the Speed Limit sign, Ulysses also updates the speed limit state variable. The speed limit state variable starts with a human-supplied value because the model is not capable of interpreting the general environment accurately.

The final acceleration constraint is generated by traffic in front of the robot. The safety goal implicit to Ulysses requires that an adequate headway be maintained to the car in front of the robot in the same lane. Ulysses therefore monitors the next car ahead in the corridor. If the lead car were to suddenly brake, it would come to a stop some distance ahead. This is the constraint point for computing an acceleration limit. The program uses the lead car's speed, its distance, and its assumed maximum deceleration rate to determine where it would stop. This activity is commonly called "car following."

After all acceleration constraints have been generated, they are combined by taking their logical intersection. The intersection operation guarantees that all constraints are met simultaneously. Figure 3-4 illustrates this process. Since the constraints in effect allow a range of accelerations between the negative limit (the braking capability of the robot) and a computed positive value, the intersection results in a range from the negative limit to the smallest computed positive value. Ulysses chooses the largest allowed acceleration in order to further the implicit goal of getting to the destination as quickly as possible. In effect, Ulysses uses *preferences* that choose higher accelerations over lower ones.

The perception system may not be able to detect objects beyond some range. Ulysses deals with the resulting uncertainty by making two assumptions: first, that there is a known range within which perception is certain; and second, that objects and conditions that trigger constraints are always present just beyond this range. This latter assumption is the worst case. For example, Ulysses always assumes that the road ends just beyond road-detection

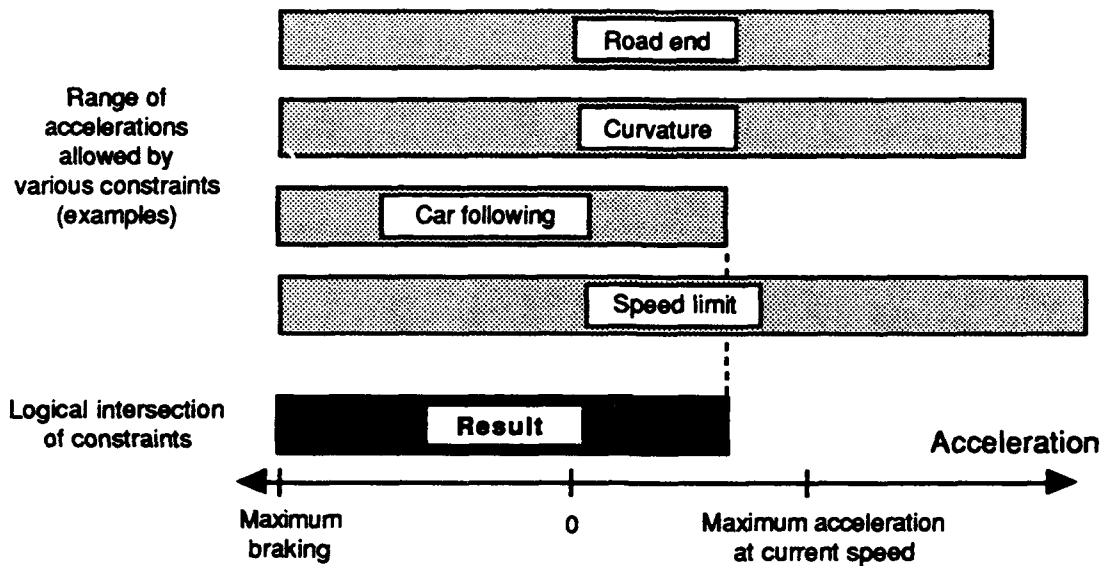


Figure 3-4: Combining acceleration constraints.

range, that the speed limit drops to zero just outside of sign-detection range, and that there is a stopped vehicle just beyond car-detection range. In this way Ulysses prevents the robot from "over-driving" its sensors.

3.3.2. An Intersection Without Traffic

The next traffic scenario we consider is a simple intersection with only one lane on the robot's approach, no other traffic, and no pedestrians. Figure 3-5 illustrates this scenario. The perception system detects intersections when the road branches or the markings on the lane ahead end. Other clues, such as traffic signals or signs (or cars, in a different scenario), may be detectable at longer ranges; however, since the robot is constrained to stop anyway at the end of the detected road, Ulysses does not consider these clues. Future versions of Ulysses may model the uncertainties of machine perception in more detail and use several clues to confirm observations.

When the robot is actually in an intersection, Ulysses can no longer detect a lane directly in front of the robot. When this first happens, Ulysses changes the "In Intersection" state variable from Street to Intersection. Later, when there is a lane in front of the robot, the state is changed back again. These state changes mark the robot's progress in its strategic route plan.

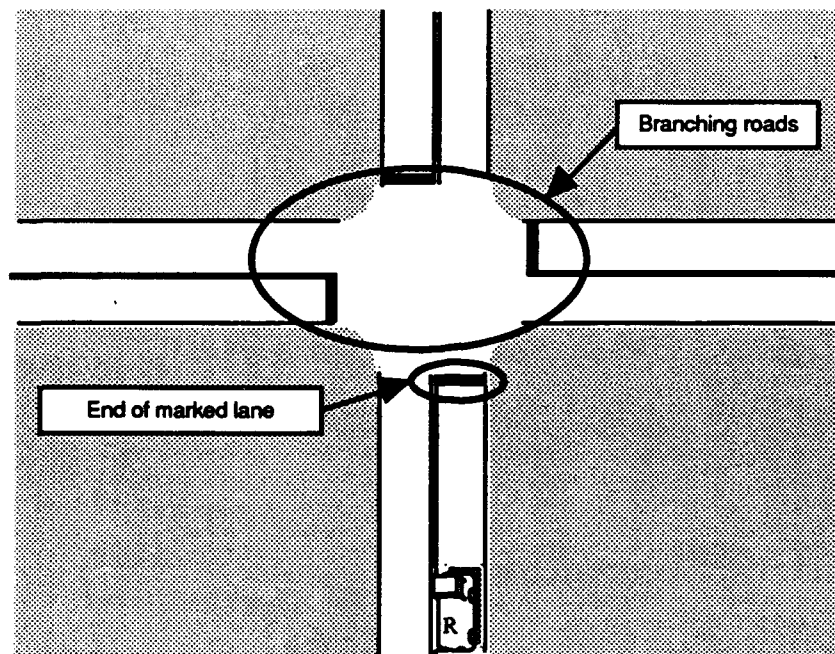


Figure 3-5: An intersection without traffic; detection by tracking lanes.

Finding the corridor is more difficult at an intersection than it is on a simple highway because the corridor no longer follows clearly marked lanes. The tactical perception system must recognize the other roads at the intersection and identify the one on which the robot's route leaves. Figure 3-6 shows that the perception system must identify the "left" road if the strategic route plan requires the robot to turn left at this intersection. Ulysses can then extend the corridor by creating a path through the intersection from the end of the robot's approach road to the start of the proper lane in the road on the left.

With the corridor established through the intersection, Ulysses generates the same acceleration constraints as for the highway case. Figure 3-7 shows several examples of how these constraints may apply at an intersection. Constraints may apply to conditions before, within, and beyond the intersection.

The next task for the driving program is to determine the traffic control at the intersection. In this scenario the only important TCD's are traffic signals, stop signs and stop markings. Thus, as shown in Figure 3-8, Ulysses requests the perception system to look for signs and markings just before intersection and signal heads at various places around the intersection.

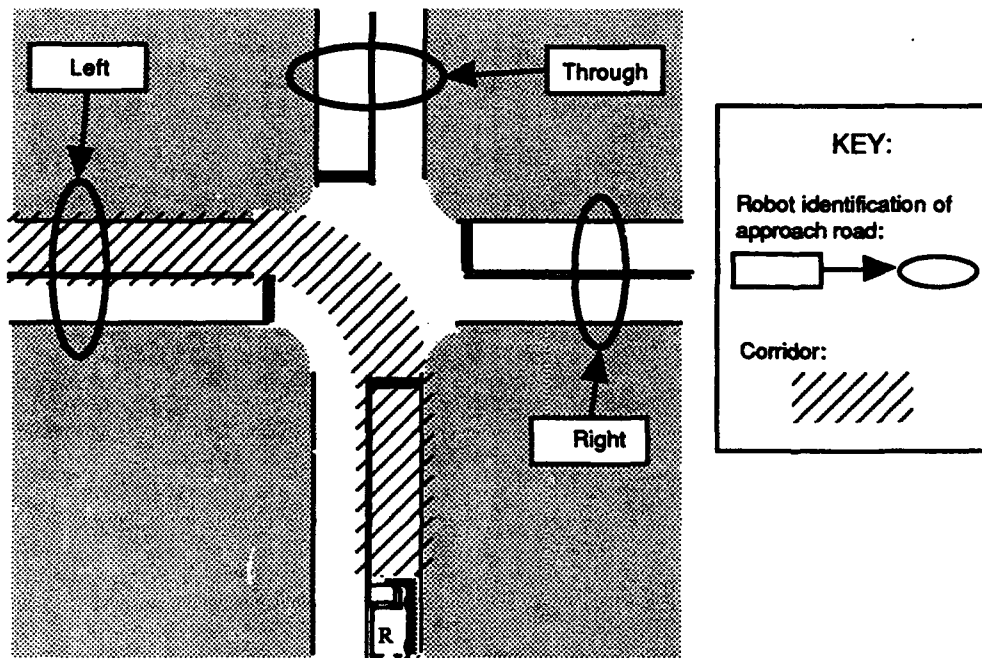


Figure 3-6: Finding the corridor for a left turn.

When the intersection is beyond the given detection range for these objects, Ulysses assumes the worst and constrains the robot to stop at the intersection; however, within the detection range Ulysses assumes that all existing traffic control devices will be found.

Figure 3-9 diagrams the decision process for traffic signals required at this simple intersection. First, Ulysses must determine what the signal indication is. If there is an arrow in the direction of the corridor, the program takes this as the effective signal; otherwise, the illuminated solid signal is used. If the signal indication is red, Ulysses generates an acceleration constraint that stops the robot at the entrance to the intersection. No constraint is generated if the signal is green. If the signal is yellow, then Ulysses determines whether the robot can be stopped at the intersection using a reasonable braking rate. If so, then the program generates a constraint as if the light were red.

Stop signs require not only speed constraints, but a short sequence of actions. A vehicle approaching a stop sign must come to a complete stop, look for traffic, and then proceed when the way is clear. Figure 3-10 shows how Ulysses performs these actions with a "Wait" state variable. When a stop sign is detected, Ulysses generates a constraint to stop the robot just before the sign. Later, Ulysses finds that the robot is stopped and that it is very near the sign, so it changes to a Wait for gap state. In this scenario there is no traffic, so Ulysses immediately moves on to the Accept state and releases the robot from this traffic control constraint.

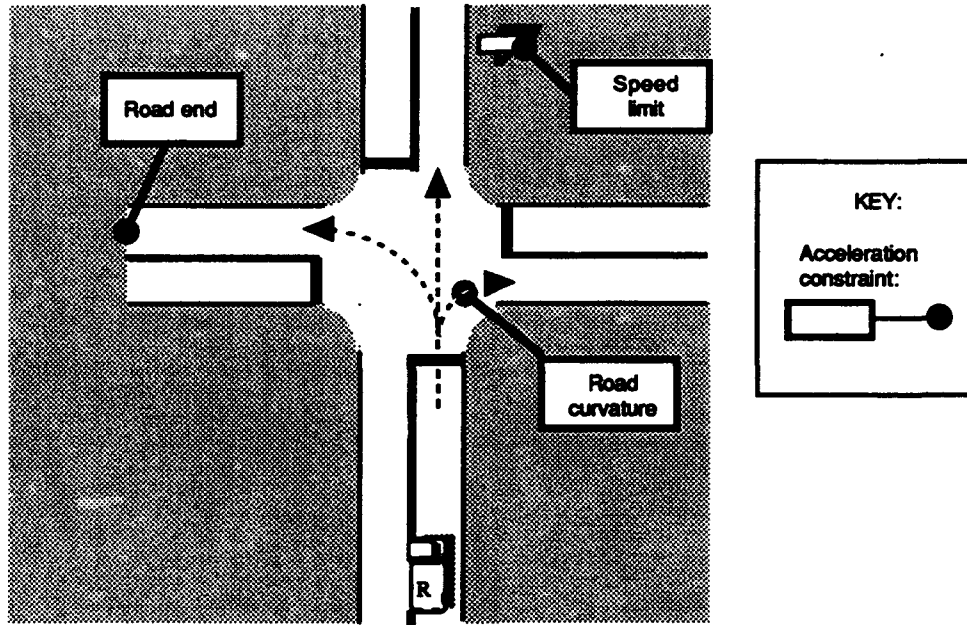


Figure 3-7: Examples of acceleration constraints from road features at an intersection.

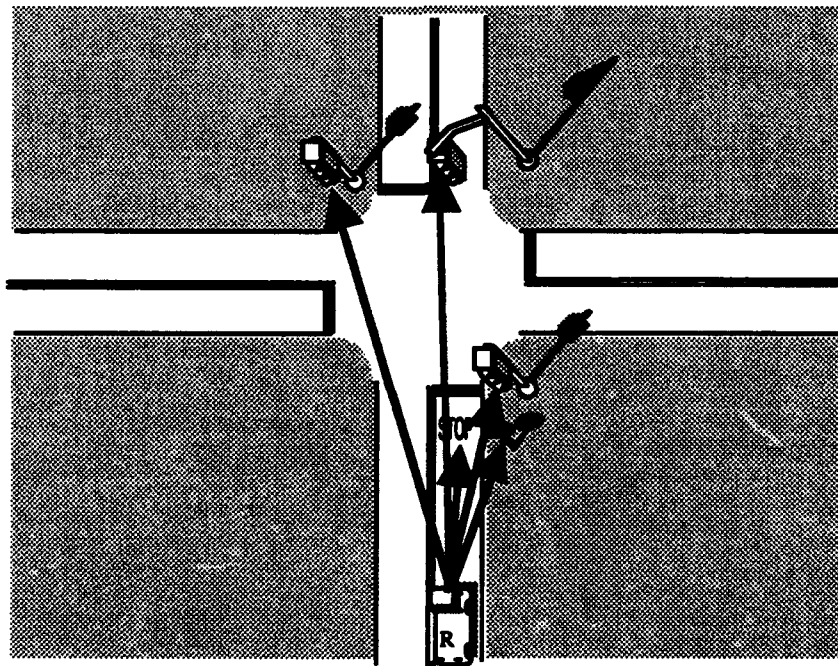


Figure 3-8: Looking for possible traffic control devices at an intersection.

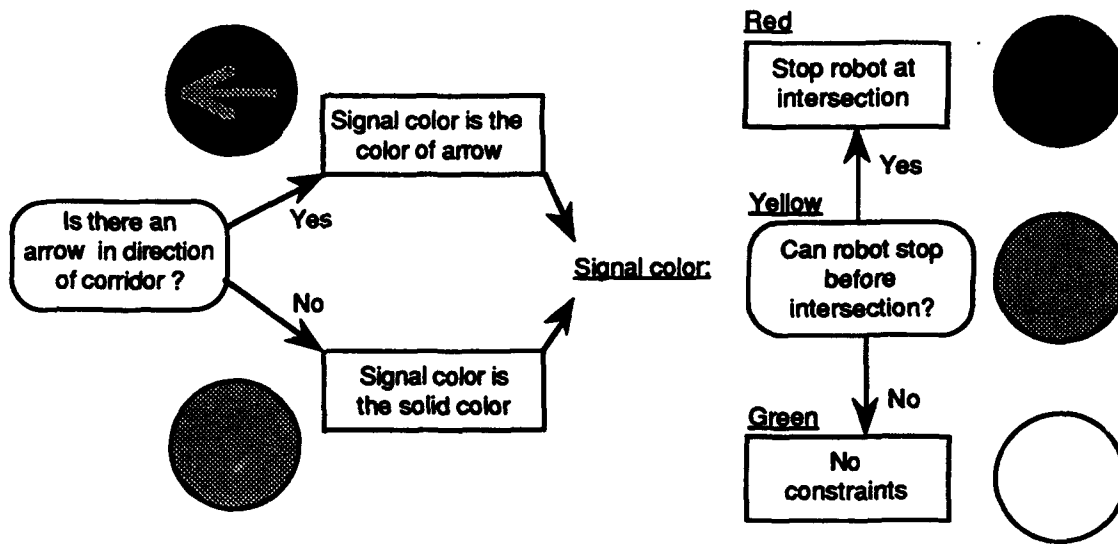


Figure 3-9: Traffic signal logic for intersection without traffic.

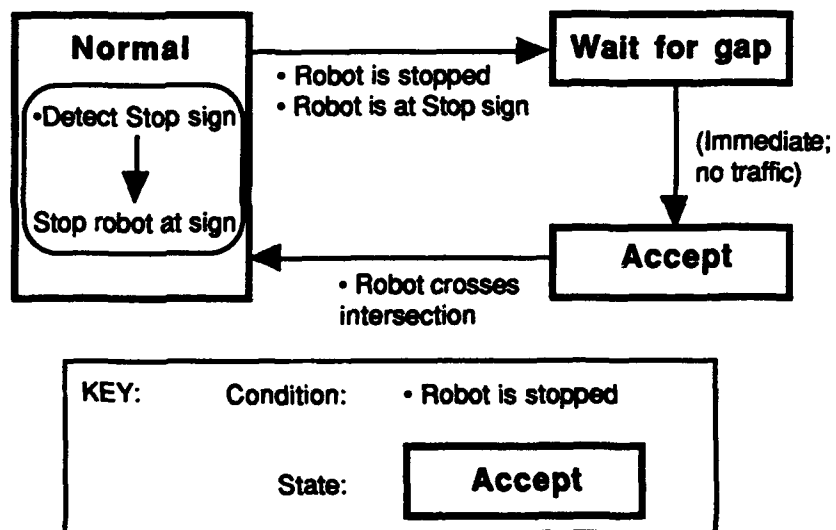


Figure 3-10: State transitions in stop sign logic.

3.3.3. An Intersection with Traffic

We will now add traffic to the simple intersection scenario. Ulysses considers three kinds of traffic at intersections: cars in the same corridor as the robot, which must be given headway; cross traffic blocking the intersection; and cars approaching the intersection on other roads which may have to be given the RoW. Human drivers are sometimes taught to watch the traffic behind them, in case another car is following too closely or approaching very fast. In such a case the driver could accept lower gaps or make more aggressive judgements at an intersection, thereby continuing through the intersection and not stopping quickly in front of the following car. Ulysses does not make such judgements, and thus does not look for traffic behind the robot.

Figure 3-11 illustrates the first kind of traffic that Ulysses considers. The driving program

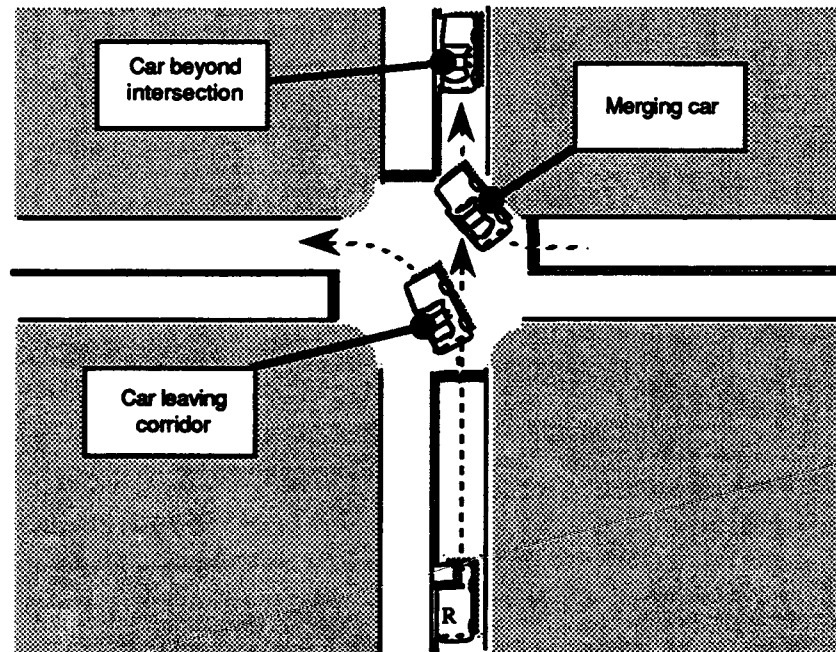


Figure 3-11: Potential car following conditions at an intersection.

must generate a car-following constraint from a lead car either in or beyond the intersection. This constraint is computed from the measured speed and range of the lead car, as before. If the lead car is turning off the robot's path, the robot must still maintain a safe headway until that car is clear of the path. Similarly, the robot must maintain a safe distance to cars that merge into its path.

The second kind of traffic that Ulysses looks for is cross traffic in the intersection. This

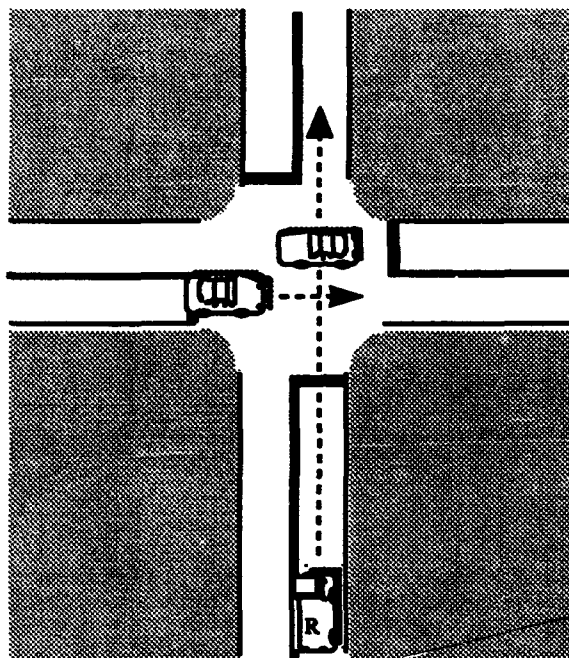


Figure 3-12: Cars blocking an intersection.

includes cars sitting across the robot's path or about to cross it, as shown in Figure 3-12. Since the law requires that a vehicle yield the RoW to vehicles already in the intersection, Ulysses looks for such vehicles and stops the robot before the intersection if it finds one. Ulysses does not currently search beyond the closest car to find a gap. Future extensions to Ulysses may try to time the robot's arrival at an intersection to the presence of an upstream gap. However, this type of behavior could only be allowed if it were safe—that is, if the robot could still stop at the intersection if traffic changed unexpectedly.

The third type of traffic is traffic approaching the intersection on other roads. The driving program analyzes nearby TCD's and this approaching traffic to determine if the robot has the RoW. After RoW is determined, Ulysses may generate an acceleration constraint and change the Wait state of the robot. Figure 3-13 illustrates this process.

Traffic Control Devices. First, Ulysses analyzes signs, markings and traffic lights. In addition to the TCD's needed for the previous scenario, the program now must consider Yield signs and the number of lanes in the roads. The lane count is used to determine if the robot is on a "Minor Road"—that is, if the robot's road has only one or two lanes and the other road has more lanes. In the United States traffic on a Minor Road generally yields the RoW to other traffic, so in effect has a yield sign. This is a practical rule, not a legal one.

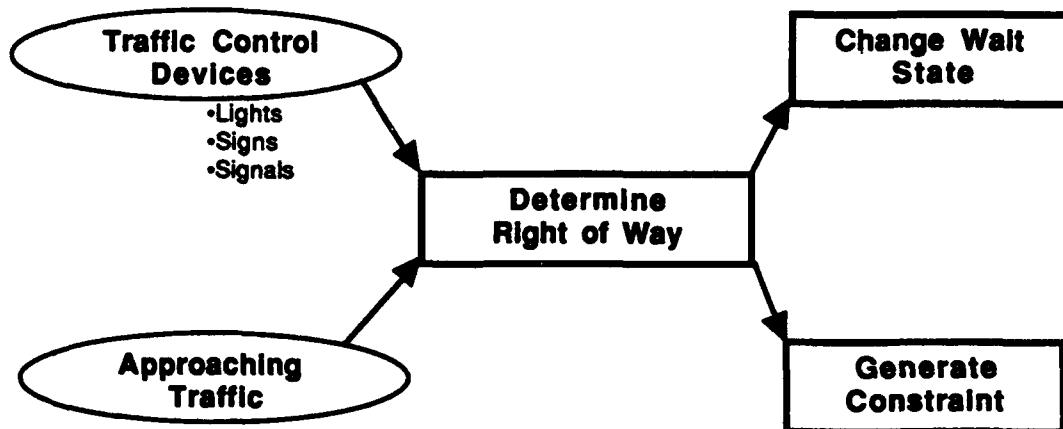


Figure 3-13: Generation of constraints from traffic and traffic control.

Table 3-1 summarizes the meanings of the various TCD's. The figure groups the TCD's into four equivalence classes. Yellow signals are treated as either red or green, depending on the robot's state and speed. If the robot is in a wait for gap state, a yellow signal is always treated as green. This practical rule allows the robot to proceed through an intersection just before the signal turns red. This allowance prevents the robot from being stuck forever in heavy traffic with no gaps. Cars with a given TCD normally yield the RoW to traffic with a TCD in a higher priority class, as will be explained below. Note that some actions in the figure do not depend on other cars; for example, red lights always stop the robot.

Approaching traffic. The second part of RoW evaluation is analyzing the robot's situation relative to other cars. This situation depends on the position and speed of the robot and other cars, and the relative position of the approach roads. Figure 3-14 shows how this information is combined with traffic control information to decide the RoW with respect to each car. For each approach, the tactical perception system looks up the road to find the first car in each lane. Perception provides the speed and distance to the intersection for each such car. The road to the right is ignored if the robot is turning right, as is the directly opposing road unless the robot or approaching car are turning left. Looking "up the road" essentially requires the perception system to find a corridor along a different road. If the perception system does not find a car, Ulysses makes RoW decisions as if there were a car at the range limit of the sensors or of sight distance. This hypothetical car is assumed to be going a little faster than the prevailing speed of traffic. If the approaching car is going too fast to stop before the intersection (assuming a nominal braking rate), Ulysses always yields the RoW.

Traffic Control	Action
Green signal OR Nothing	Use RoW rules (see Figure 3-14)
Yield sign, OR Nothing and on Minor Rd	Yield to traffic with Green or no traffic control; otherwise use RoW rules (see Figure 3-14).
Stop sign, OR turning right at Red signal when allowed.	Stop at intersection; then proceed using RoW rules (see Figure 3-14)
Red signal	Stop at intersection

Increasing priority



Yellow signal	IF Wait-state is "normal" AND robot can stop at intersection THEN treat as Red signal ELSE treat as Green signal.
---------------	---

Table 3-1: Actions required by four classes of Traffic Control Devices.

On the other hand, if the car is very far away from the intersection, Ulysses will ignore it. "Very far away" means that the time it will take the robot to cross the intersection at its current speed is less than the time it will take for the other car to reach the intersection.

If none of the above conditions exist, then Ulysses further analyzes RoW based on traffic control devices and the movements of the other cars. First, Ulysses guesses the traffic control for the approaching car. To do this, the perception system must look at the side of the approach road for a sign (the back side of a sign). If there is no sign, and the robot is facing a traffic light, Ulysses assumes that the approaching car also has a traffic light. If the car is approaching from a cross street, the light is assumed to be red (when the robot's is green), and otherwise green.² Once the program has an estimate of the approach car's traffic

²This is a particularly clear example of how the explicit, computational rules in Ulysses elucidate the information requirements of driving. Drivers must make assumptions about the TCD's controlling other cars, and sometimes make errors if their assumptions are wrong.

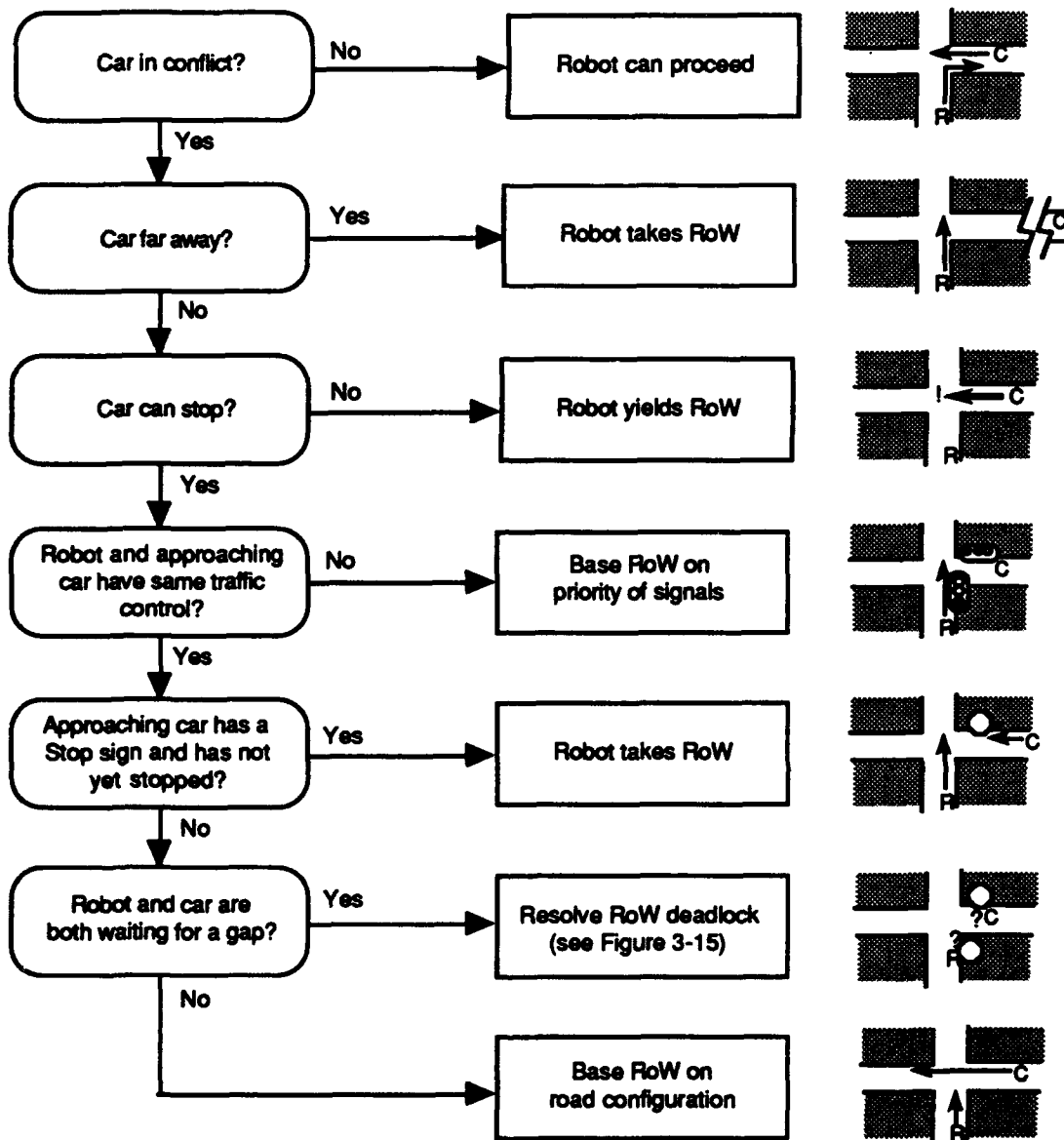


Figure 3-14: Right of Way decision process.
 'R' is the robot, 'C' is the other car.

control, it compares the control to that facing the robot. If they are not equivalent, then the vehicle with the lowest priority signal is expected to yield the RoW.

If the traffic control for the approaching car and the robot are equivalent, Ulysses performs further analysis of the situation. If the approaching car is close to the intersection and stopped, the program assumes that it is waiting for a gap in traffic. In the case of a stop

sign, the robot takes the RoW if the approaching car is not yet waiting. Otherwise, if one or both vehicles are moving, then the robot yields the RoW to cars on the right, and to cars ahead when the robot is turning left. If both the robot and the approaching car are stopped and waiting for a gap, then the drivers must use some deadlock resolution scheme to decide who goes first. In human drivers we have identified four such schemes, as illustrated in Figure 3-15. Humans use different schemes, depending on the local custom and the driver. Ulysses bases its decision only on road configuration; it will use the first-arrive, first-leave technique as well when we better understand how to recognize the same cars in images taken at different times.

After determining the requirements of the traffic control devices and evaluating the traffic situation, Ulysses may create an acceleration constraint and update the wait state. Figure 3-16 shows the conditions for changing states and for constraining the robot to stop at the intersection. Note that once the program has decided to accept a gap and take RoW, only the presence of new cars in the robot's path will cause it to stop again.

3.3.4. A Multi-lane Intersection Approach

The driving situations considered so far have not included streets with multiple lanes. With this complication, Ulysses must generate lane use constraints and make lane selections. We first consider the simple case of multiple lanes at an intersection, as depicted in Figure 3-17. In this scenario there is no other traffic on the same road as the robot.

As the robot approaches an intersection, constraints on turn maneuvers from different lanes—which we generally refer to as lane *channelization*—determine the lane-changing actions that Ulysses can take. Ulysses determines if the channelization of the robot's current lane is appropriate, and looks for a better lane if it is not. Channelization is estimated first by finding the position of the current lane at the intersection. If it is the left-most lane, then the program assumes that a left turn is allowed from the lane; similarly with right turns if it is the right-most lane. Through maneuvers are allowed from any lane by default. Ulysses also looks for signs and road markings to modify its assumptions about channelization.

If the robot's intended maneuver at the intersection is not allowed by the channelization of the robot's current lane, Ulysses generates an acceleration constraint to stop the robot before the intersection. Figure 3-18a illustrates this constraint. Ulysses estimates the minimum distance required to change lanes (from the width of the lanes and the robot's minimum turn radius) and stops the robot that far from the intersection. The program next decides in which direction the robot should move to correct the situation, and looks to see if there is a lane on that side of the robot. If there is a lane, Ulysses constrains the robot to take a lane-change action in that direction. As a practical matter, solid lane lines are ignored,

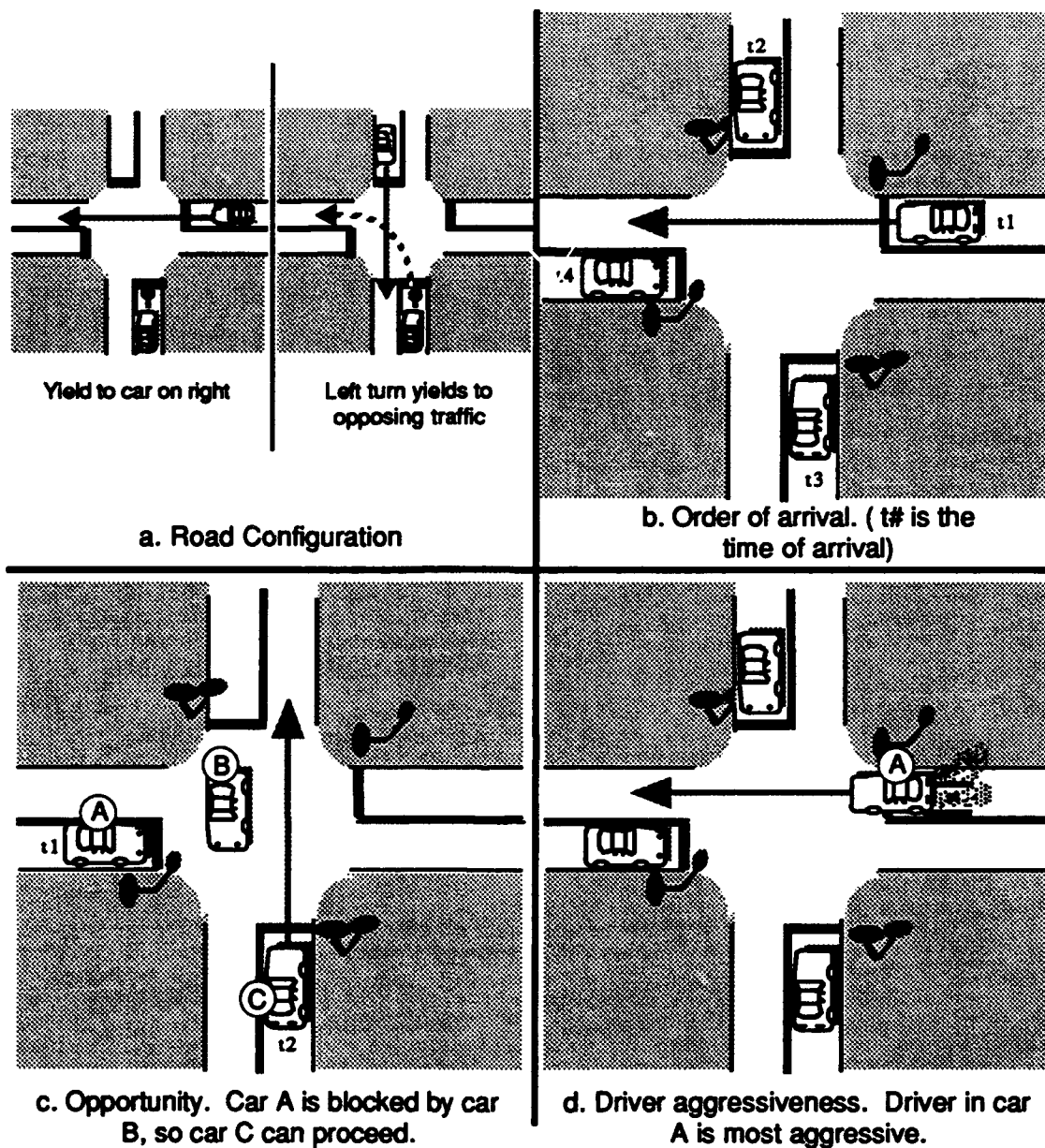


Figure 3-15: Right of Way deadlock resolution schemes.

because making a turn from the correct lane is more important than strictly obeying these lane-change restrictions. Since there is only one allowed action, no selection preferences need be considered.

Once a lane-changing action is started, Ulysses makes sure that the robot completes the

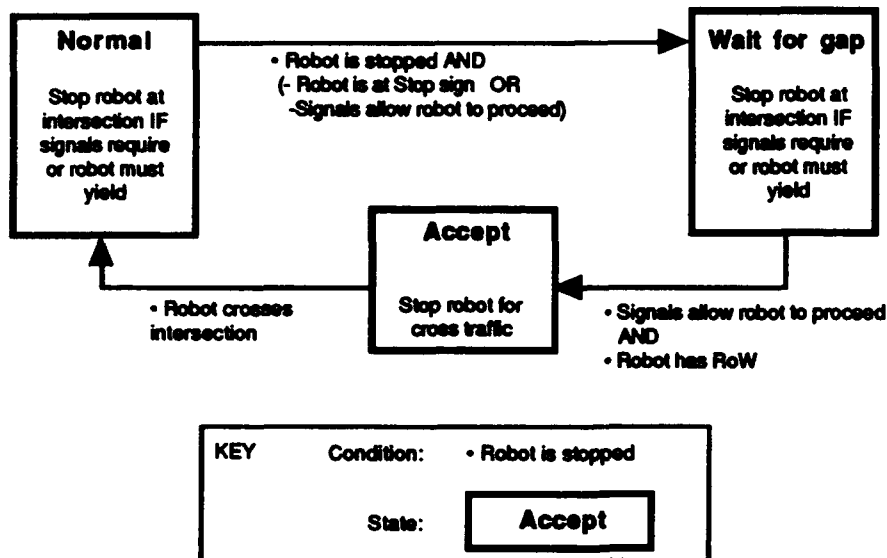


Figure 3-16: Changes in the Wait state at an intersection.

maneuver before it enters the intersection. The program does this by generating an acceleration constraint that slows the robot to a low speed at the minimum lane changing distance described above. Figure 3-18b shows this constraint. The low speed is set to keep the lateral acceleration on the robot below a certain threshold when turning with minimum radius. The lane-changing constraint is conservative, but our model of operational control does not provide Ulysses with enough information about the lane changing maneuver to predict just when it will be complete (see description of operational level, below). By driving at a speed low enough to use the minimum turn radius while far enough from the intersection, Ulysses guarantees that the maneuver can be completed.

With multiple lanes the driving program recognizes that there may be different signal indications for different lanes. The perception system looks across the intersection from right to left and attempts to find all of the signal heads. If the robot is turning, Ulysses uses the indication on the head on the side of the turn. Otherwise, Ulysses uses the head most nearly over the robot's lane.

When Ulysses decides to initiate a lane change, it sets a "Lane Position" state variable to Init-Left or Init-Right appropriately. Figure 3-19 shows these state transitions. The program does not consider lane selection in future decision cycles while in one of these states. This state variable also helps to guide the perception functions when the robot is between lanes. Once a lane change is initiated, the operational control subsystem moves the vehicle

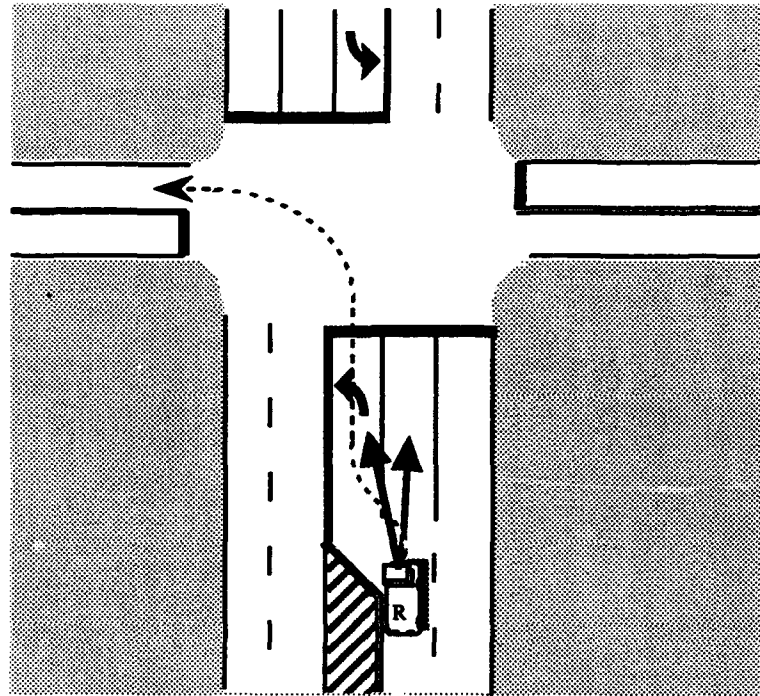


Figure 3-17: Robot observation of lane positions and markings at a multiple lane intersection approach.

all of the way to the new lane. When the robot is straddling a lane line, the state changes to Changing Left (Right). When tactical perception no longer sees the robot between lanes, the state reverts to Follow Lane.

3.3.5. A Multi-lane Road with Traffic

The next scenario is a highway far from an intersection. Unlike the first scenario above, this one is complicated by multiple lanes. Figure 3-20 shows some of the new considerations involved. First, a lane may end even without an intersection. Perception detects this in much the same way as it detects a complete road end, and Ulysses creates an acceleration constraint to stop the robot before the end of the lane. This constraint disappears after the robot changes lanes. Multiple lanes also make car following more complex. If the car in front of the robot is between two lanes, Ulysses maintains a safe headway to that car but also looks for another car in the lane. If the robot is itself changing lanes, the program looks ahead for cars in both lanes until the robot has cleared the old lane.

The three lane actions available to the robot are following the current lane, moving to the

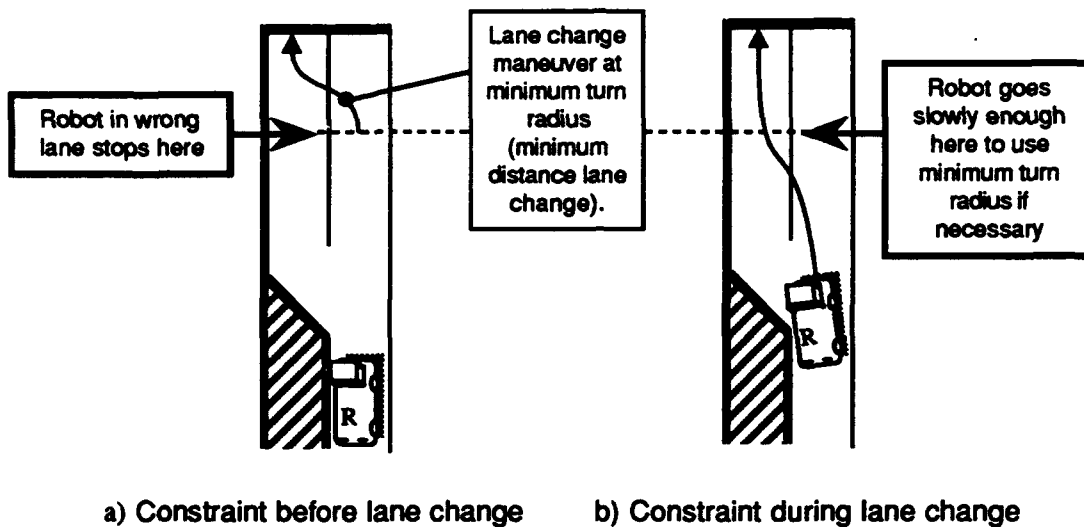


Figure 3-18: Channelization acceleration constraints.

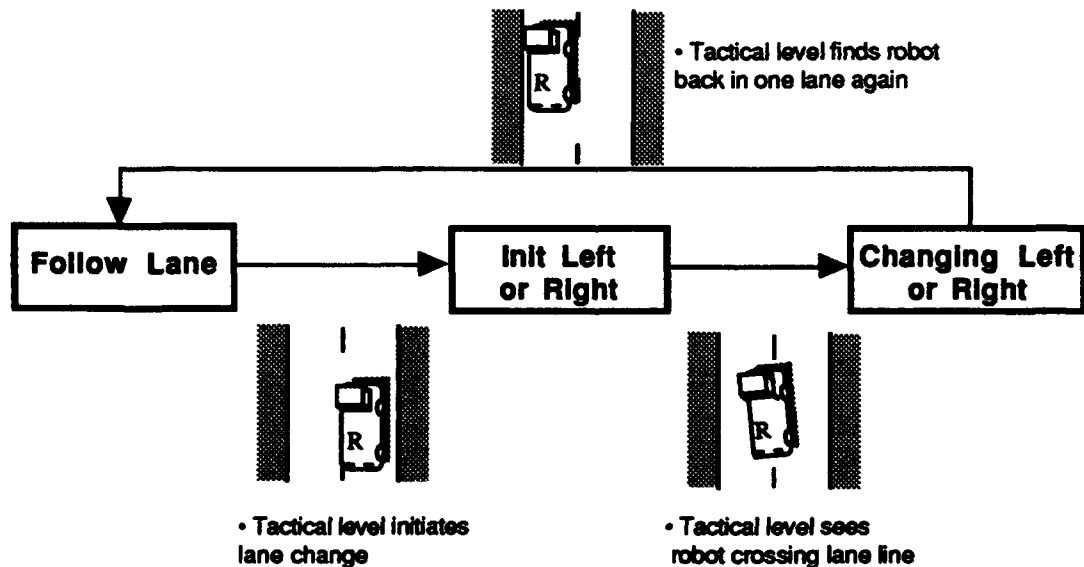


Figure 3-19: State changes during lane changing.

right, or or moving to the left. Moving to the left does not currently include moving into the opposing lanes; thus Ulysses cannot yet overtake when there is only a single lane in each direction. Ulysses generates constraints and preferences for changing lanes if certain traffic conditions exist. Figure 3-21 shows the important conditions: adjacent lanes, gaps in traffic in adjacent lanes, blocking traffic in the robot's lane and blocking traffic in adjacent lanes.

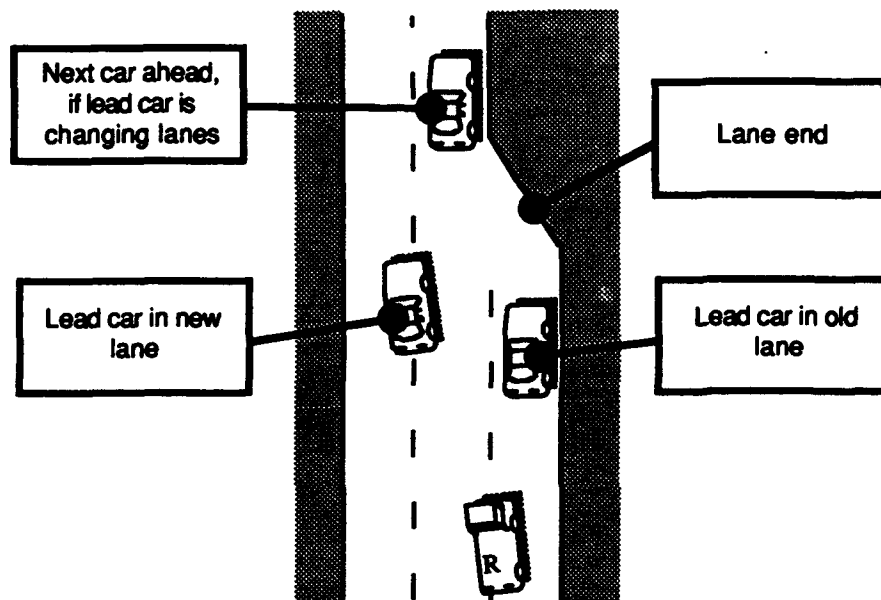


Figure 3-20: Multiple lane highway scenario.

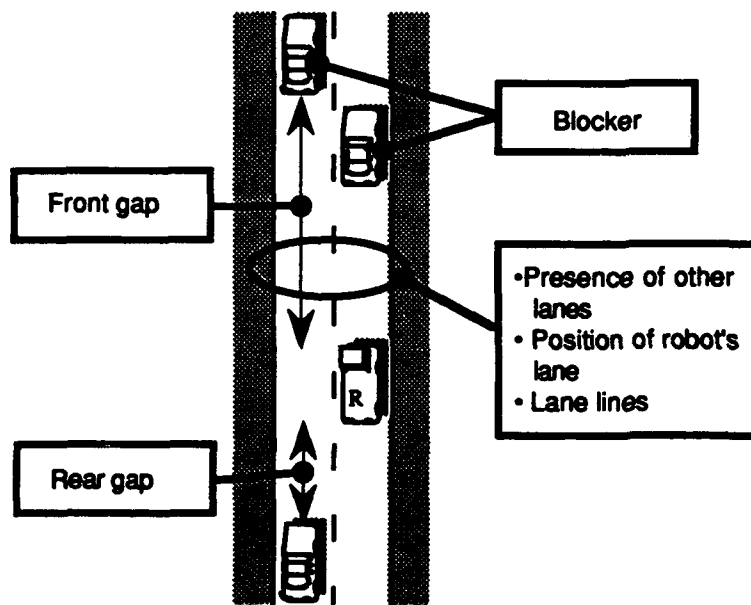



Figure 3-21: Traffic objects that affect lane changing decisions.

Table 3-2 shows the constraints and preferences for each traffic condition. The absence of adjacent lanes or broken lane lines or gaps in traffic eliminates the option to move to adjacent lanes. Ulysses generally prefers the rightmost lane to others. Traffic *blocks* the

CONDITION	SET OF ALLOWED ACTIONS	PREFERRED ACTION	Increasing Priority 
Blocking car (Robot blocked by traffic in lane*)	• $L - C > T_1$ Broken lane line	All Move Left	
	• $R - C > T_1$ Broken lane line	All Move Right	
	• $C > R, L$	All Keep Same	
Gap for merging			
	• No gap in traffic to left	(Move Right, Keep Same) —	
	• No gap in traffic to right	(Move Left, Keep Same) —	
Available lanes			
	• There is no lane to the left	(Move Right, Keep Same) —	
	• There is no lane to the right	(Move Left, Keep Same) —	
	• (There is a lane to the right) AND $(R \geq C)$	All Move Right	
Default	All	Keep Same	

KEY	<ul style="list-style-type: none"> *: True when (Car following accel) < (Other accel constraints) - T_2 T_i: Arbitrary threshold value. L, R, C: Max. acceleration allowed in the lane to the Left, lane to the Right, and Current lane, respectively
-----	---

Table 3-2: Lane action preferences for a highway with traffic.

robot if the acceleration allowed by car following is significantly less than acceleration allowed by other constraints. Blocking traffic creates a preference for an adjacent lane if the acceleration constraint in that lane is significantly higher than the car following constraint in the current lane. Ulysses estimates the allowed acceleration in the adjacent lane by hypothesizing what the car following constraint would be and combining this with the speed limit, lateral acceleration and road end constraints. The program combines the constraints from various conditions by taking their logical intersection—retaining only actions that appear in all constraints. If there is more than one lane action available after constraints

have been combined, preferences determine which action is chosen. Preferences due to blocking traffic are given priority over the rightmost lane preference.

Ulysses judges gaps for changing lanes based on two criteria: first, the deceleration required of the robot to create a safe headway to the lead car in the other lane; and second, the deceleration required of the following car to leave space for the robot. The system does not search for a gap farther back in traffic if the adjacent one is not big enough. However, if the robot is blocked, traffic in the adjacent lane will tend to pass the robot, thereby providing new merging opportunities. Ulysses also does not have the ability to use a physical "language" (beyond directional signals) to communicate to other drivers that it is anxious to change lanes. It is thus possible that the robot will never find a gap if traffic is congested³.

3.3.6. Traffic on Multi-lane Intersection Approach

The next scenario combines the previous two multi-lane cases. This situation, illustrated in Figure 3-22, adds channelization constraints to traffic considerations. When the adjacent

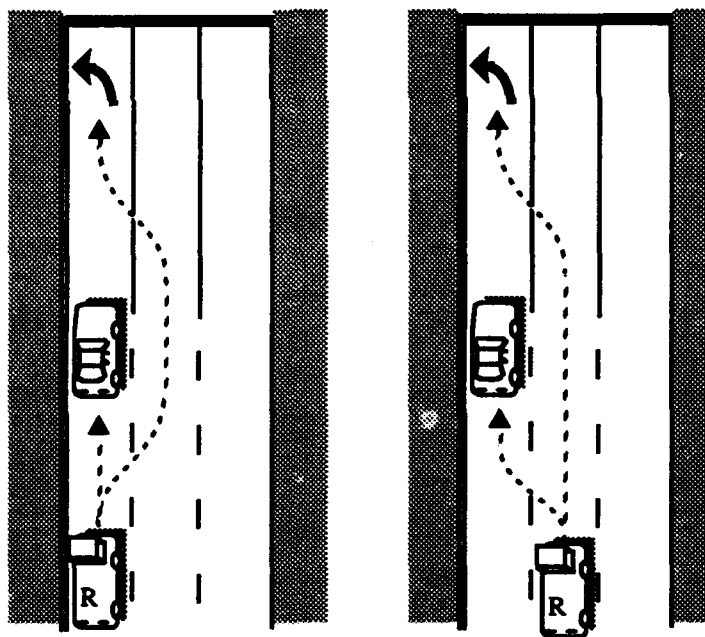


Figure 3-22: Decisions at a multiple lane intersection approach with traffic.

lane does not permit the robot's next turn, Ulysses must decide whether a pass can or should

³That is, without allowing an unsafe headway—a condition in which a deceleration by the robot or the lead car could cause an accident

be made. Similarly, if the robot needs to move to a turn lane that has traffic in it, Ulysses must decide whether to merge immediately or wait. Table 3-3 lists the constraints and preferences that must be added to those in Table 3-2. When combining preferences, Ulysses

CONDITION	SET OF ALLOWED ACTIONS	PREFERRED ACTION	PRIORITY
Channelization			
• $d < D_{Min}$	Keep Same	—	Highest
• $D_{Min} < d < D_{Far}$	Current lane OK (Keep Same, Move to <OK lane>)	—	
	Current lane not OK (Keep Same, Move to <OK lane>)	Move to <OK Lane>	
• $d > D_{Far}$	All	Move to <OK Lane>	Just below Blocking Car in Table 3-3
• No intersection visible	All	—	—

KEY

d : Distance from robot to intersection

D_{Min} : Minimum distance needed to change lanes.

D_{Far} : Range of influence of intersection on upstream traffic.

OK: Channelization allows (or is closest to) robot's intended route

Table 3-3: Lane action constraints and preferences at an intersection with traffic.

gives the channelization demands the highest priority unless the robot is far from the intersection.

When the robot is near the intersection ($d < D_{Far}$ in Table 3-3), it is not allowed to change lanes to move away from an acceptable ("OK") lane. Thus Ulysses will not attempt to pass a blocking car near an intersection unless multiple lanes are acceptable. This restriction may cause the robot to get stuck behind a parked vehicle, but it avoids the problem of deciding whether the lead vehicle is really blocking the road or just joining a queue that extends all the way to the intersection.

The "Far from intersection" ($d > D_{Far}$) condition in Table 3-3 depends on a distance threshold D_{Far} . When the intersection is at least D_{Far} away from the robot, Ulysses can pass blockers rather than staying in a turn lane. The threshold distance is properly a function of traffic—how far traffic is backing up from the intersection, whether there are gaps in the

traffic ahead of the blocker, and how long (distance) the robot would take to complete a pass. Ulysses determines D_{Far} by estimating passing distance and adding a margin to cover the other conditions. The margin is a function of the speed limit, but does not explicitly incorporate downstream queuing conditions. In our simulated world we have found that the margin needed to minimize inappropriate passing maneuvers puts the robot in the "near intersection" condition soon after it detects the intersection.

The presence of traffic introduces additional factors into the tests of blocking conditions. When Ulysses determines what acceleration is possible for the robot in an adjacent lane, it must consider constraints from lane channelization and intersection traffic control. In effect, Ulysses must hypothesize the robot in the adjacent lane, look ahead in a new corridor, and recompute all of the acceleration constraints from that lane.

When the robot wishes to change lanes because of channelization, but cannot because that lane change action is not allowed (for example, because a gap in traffic is not available), Ulysses changes the robot's Wait state to "wait for merge gap." This action signals a standing desire to change lanes. When the robot is in this state, Ulysses adjusts the calculation of the car-following constraint so that extra space is left in front of the robot for maneuvering.

3.3.7. Closely spaced intersections

The final complication comes with multiple intersections spaced closely enough together that they all affect the robot. Figure 3-23 depicts multiple intersections ahead of the robot and on cross streets. The driving knowledge already described is sufficient to get the robot through such a situation. However, this scenario illustrates the necessity of tracing a corridor ahead through multiple intersections, analyzing traffic and traffic control at every intersection along the corridor, and looking through intersections on cross streets for approaching cars.

3.4. Interface to the World

3.4.1. Tactical Perception

Tactical driving decisions require information about roads, intersections, lanes, paths through intersections, lane lines, road markings, signs, signals, and cars. Our driving model assumes that the robot has a perception subsystem that can detect these traffic objects and determine their location and velocity. In addition, the perception system must estimate the distance to objects, the distance between objects, and the velocity of cars. There is more perception at the operational and strategic levels, but we do not address it in this model.

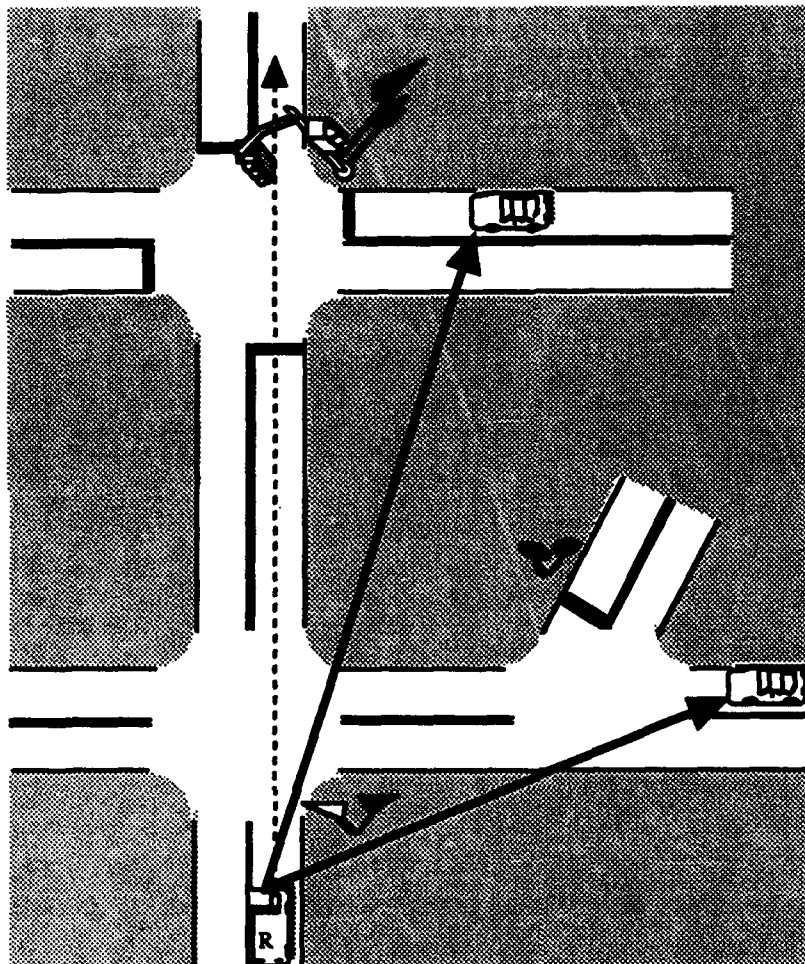


Figure 3-23: Visual search through multiple intersections

Tactical driving requires information about spatial relations between objects. When Ulysses looks for an object, it is really interested in objects that have a specific relation to another object. Figure 3-23 illustrates this concept. In the figure, Ulysses must at some point in its analysis look for cars on the *right-hand approach road* at the *second intersection* ahead. The model assumes that these type of spatial relations can be tested by the perceptual system. Chapter 5 discusses techniques for detecting these spatial relations.

3.4.2. Modeling Time

Ulysses specifies what the robot should do at any moment, given what it observes in the world. In developing the model I have found that it is possible to interpret a driving scene almost completely from current observations, i.e. with little state information. In addition, Ulysses does not project the current situation into the future to predict the outcome of different actions. Ulysses is thus a model of a nearly purely reactive agent. The state variables Ulysses uses are summarized in Table 3-4; except for the Speed-limit variable, these few bits of state are mostly used to provide hysteresis to eliminate uncertainties around measurement thresholds. Ulysses generates constraints with the implicit knowledge that the world can be observed again in the next decision cycle. A cycle time of 100 milliseconds was found to provide adequate performance.

State Variable	Values
Speed Limit	<current speed limit>
In intersection	Street, Intersection
Wait	Normal, Wait for gap, Wait for merge gap, Accept
Lane Position	Follow lane, Init-left (-right), Changing-left (-right)

Table 3-4: Ulysses state variables.

3.5. Model Limitations

Ulysses was designed to drive safely in the PHAROS world. The goal was to prevent the simulated robot from having or causing accidents, and from unnecessarily constraining itself to stop. Ulysses achieves this goal for many traffic situations, including different intersection configurations, a variety of traffic control devices and associated rules, multiple lane roads, and freeway interchanges with light traffic. Of course, since PHAROS is a simplified world, we would not necessarily expect Ulysses to work in a more complex driving environment. This section describes some of Ulysses' limitations and discusses whether they represent fundamental limits or whether Ulysses could be extended to more realistic domains in its current form.

3.5.1. Embedded Assumptions

There are many specific assumptions about the domain embedded in the Ulysses driving model. These assumptions allow Ulysses to take advantage of the simplifications present in PHAROS. However, in many cases Ulysses could be extended in its present form to allow these assumptions to be relaxed. For example:

- **Traffic objects.** Ulysses assumes that the world is limited to the traffic objects it knows. The model could be extended to consider new objects (and the resulting new situations) by adding new constraints and preferences to the existing ones. In this way Ulysses could handle such things as pedestrians, fire trucks, new regulatory signs, unsafe tail-gaters, or mid-block traffic signals. However, there is no provision in the model for reading and understanding new signs, or for learning the semantics of new objects from experience.
- **Physical constants.** Ulysses assumes specific values for the vehicle's braking and cornering limits, other vehicle's braking limits, the delay to the next decision time, etc. These constants could be treated as variable parameters to allow Ulysses to handle other situations—for example, new braking characteristics on wet roads.
- **Lanes.** Ulysses assumes a traffic world with well defined lanes. Some of the problems with this assumption could be removed by adding maneuver commands to move to one side of the lane or another, without changing lanes. This addition would require extending the lane-position state variable and adding appropriate rules and constraints (in parallel with existing ones). Ulysses could then pass bicyclists in the right part of the lane, go around vehicles stopped in the left part of the lane while waiting to make a left turn, or move right when approached by an oncoming car on an unmarked rural or residential street.

In other cases the well-defined-lane assumption is more limiting. For example, if the autonomous vehicle were required to thread its way between cars at a congested intersection, or drive around a road repair site, or drive across a parking lot, the lane assumption would be completely invalid. In these cases Ulysses would have to replace its routine lane-driving knowledge with other routine knowledge. In other words, when various conditions were detected, Ulysses would have to switch "modes" and *ignore* some rules and *activate* different ones. This requires a bigger change to the model than merely adding new rules to existing ones.

- **Perception.** Ulysses effectively assumes that the robot has perfect perception. Thus it does not have to describe what to do if some fact is not known with certainty. Human drivers are faced with this problem all of the time because they cannot look everywhere at once, and because objects in the real world are sometimes occluded. Chapters 6 and 7 describe how the Ulysses may be implemented in such a way as to choose a safe action when it is uncertain about sensory data. In Chapter 7 I will also discuss how one might relax the assumption that safety is the most important goal.

3.5.2. Other Limitations

Other limitations prevent Ulysses from handling certain anomalous events or performing better in complex situations. Improvements to Ulysses in these cases would require structural changes. For example:

- **Detecting long term situations.** Except for the speed-limit state variable, the model has no long-term memory. Thus there is no way to detect problems that only become apparent after a period of time, e.g. a broken traffic light, heavy traffic that prevents merging or crossing, etc. Some such problems could be handled with different parameters at the tactical level, while others would require changes in the strategic plan.
- **Predicting future worlds.** Some traffic situations, for example cars in multiple lanes traveling the same direction, are especially complex and uncertain. It would be difficult to determine a rule that detected a traffic pattern and generated a constraint based on a possible problem. Predicting a problem (i.e., by modeling future world states) would be intractable if small vehicle movements were used as the incremental state change. Prediction would probably be much easier with a more abstract notion of (relative) traffic motion. Ulysses does not have any such abstraction for reasoning about situations, and so does not reason by predicting future states.
- **Changing driving styles.** The constraints and preferences in Ulysses generate a particular style of driving. The style is evident in the headways and gaps Ulysses uses, in its decisions when to change lanes, and in its behavior at intersections. The style should really be adjustable from the strategic level, based on changing goals—for example, maximizing fuel economy, minimizing trip time, obeying the letter of the law, etc.

3.6. Summary

Ulysses is a computational model of tactical driving. Driving knowledge is encoded as constraints and preferences on acceleration and lane choice. The effects of constraints and preferences depend on current conditions in the world. Ulysses obtains information about various traffic objects and computes the constraints and preferences. The relevance of particular traffic objects depends on their spatial relation to the robot and to each other. After constraints have been generated, they are intersected to determine the net constraint. Prioritized preferences then select the robot's action.

Ulysses can drive a robot in traffic on multi-lane roads, through intersections, or both. Ulysses looks for lane endings, curves, speed limit signs, and other cars to constrain the robot's acceleration on a road. When approaching an intersection, the model requests (sensory) data on signs, signals, markings, other roads, and approaching traffic as well as the general road features just mentioned. Ulysses determines the semantics of the traffic control devices and the road configuration and decides whether the robot has the right of way to enter the intersection. Additional acceleration constraints prevent the robot from entering the intersection in the wrong lane or while changing lanes. The program also evaluates lane

channelization, potential speed increases in adjacent lanes, traffic in adjacent lanes, and lane position in order to select a lane.

These capabilities make Ulysses a competent model for implementation as a robot driving system. The next chapter discusses some of the architectural issues that are relevant to implementing such a system. The remainder of the thesis discusses three implementations of Ulysses that use increasingly powerful techniques to focus perceptual attention and lower computing cost.

Chapter 4

Building a Driving System

The previous two chapters provide the background for research in driving perception. They describe what it is the robot has to do in order to be driving "correctly" at any given moment. Such an ideal model of behavior must be implemented in the context of real world dynamics—domain dynamics, robot motion, and processing delays. In addition, a tactical driving system for a real vehicle must be embedded in a system that addresses the other driving levels. While the goal of this thesis is not to design a new architecture for robot control, it is nevertheless important to establish some nominal system design parameters. Defining an architecture will clarify which system design issues are and are not addressed by Ulysses. This chapter first describes how the tactical driving model is assumed to fit with other levels of driving. Next it describes the execution control system that the Ulysses driving program uses. Finally, it shows how active vision is introduced into the system.

4.1. Integrating Driving Levels

4.1.1. Current Implementation

The Ulysses model of tactical driving is intended to fit within a system that includes all driving levels. Figure 4-1 is a schematic of how different levels are assumed to interact for the purposes of the Ulysses implementations discussed in this thesis. In this driving system, the "commands" that are passed down between layers describe *activities* or *behaviors* that the lower level performs for a period of time. This type of command allows the lower level to work at a different time scale, sense different data and perform a different kind of task—all in parallel with the higher level. Once the activities are started, the lower level does not need further intervention for a while, and can produce appropriate robot behavior even if the higher level does not issue a new command for some time.

The normal completion of commands at one level is not directly reported to higher levels in Figure 4-1; instead, the world and the perception components do this indirectly. For example, the tactical level discovers for itself that lane change and intersection traversal actions are completed; the strategic level detects when an intersection has been crossed and a route instruction has been executed.

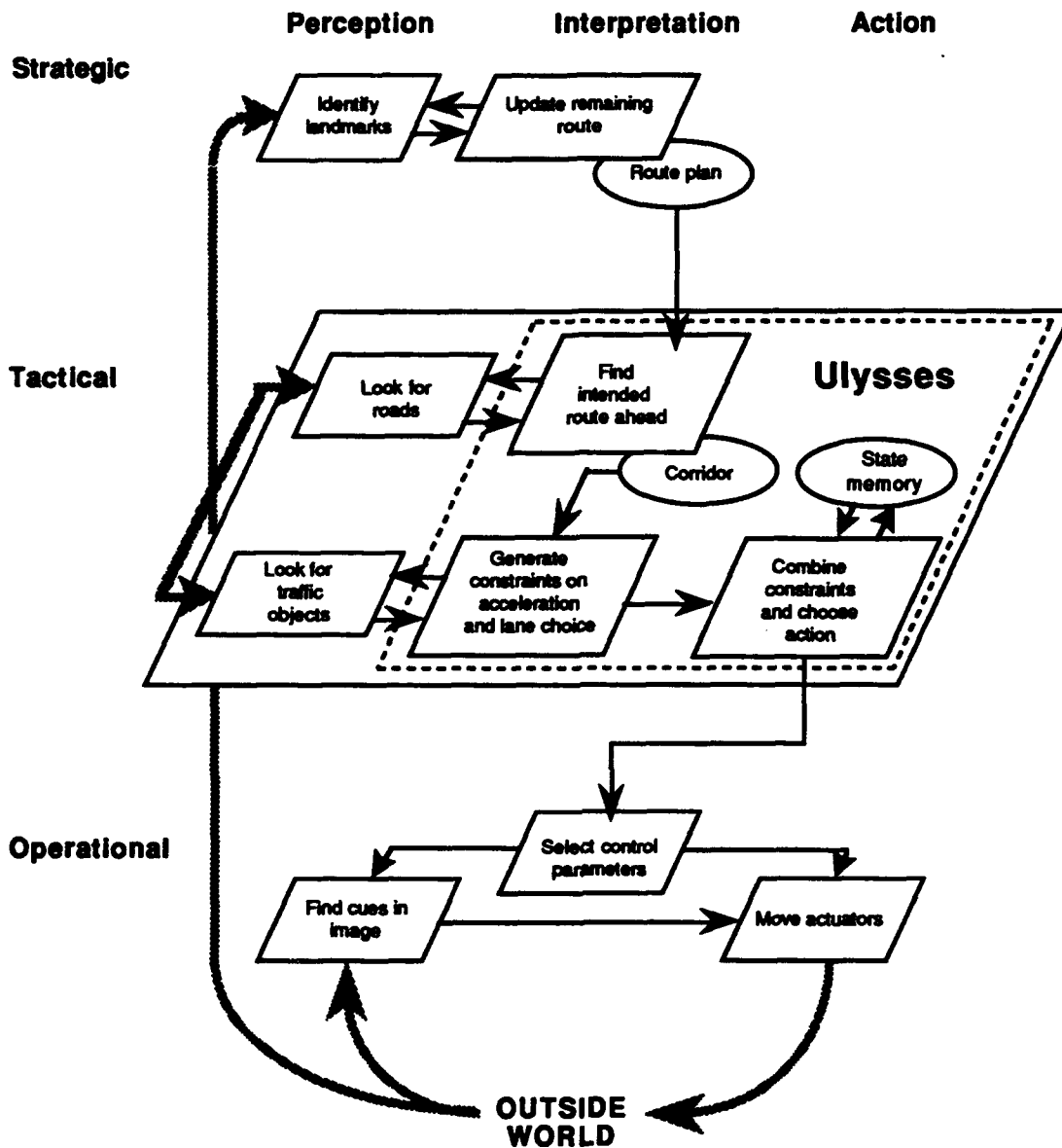


Figure 4-1: Schematic of a driver model

The strategic level generates the route plan and looks at the world when necessary to determine the robot's progress on the plan. The current driving program implementation uses the simplest possible strategic planner—routes are created by a human user and given directly to the tactical level. The plan is assumed to be based on an accurate map of the street network, so the desired maneuver at every intersection is known ahead of time.

The tactical level uses the route plan as a guide in assessing the traffic situation. The

route plan determines where the robot should look (through intersections) to create the corridor. The corridor is a data structure that records information about the visible route ahead of the robot. Tactical driving knowledge (in the form of constraints and preferences, as described earlier) uses the corridor and other sensed objects to generate maneuver commands.

The operational level makes emergency reactions and executes commands passed down from the tactical level. The operational level is assumed to have its own perception functions to detect emergencies and provide feedback to the actuator controllers. Emergency reactions include avoiding collisions with objects that move into the robot's path or avoiding potholes. These are considered emergencies because they involve non-traffic objects or anomalous behavior—i.e., they are outside the scope of the driving model. The tactical commands are presumably implemented at the operational level by selecting from a library of behaviors. These behaviors can perform four functions: maintaining an acceleration, tracking a lane, changing lanes, and traversing an intersection.

Acceleration. The operational level keeps the vehicle's acceleration at the value commanded by the tactical level. Speed control is not used because an instantaneous change in speed requires an infinite impulse of force, while a change in acceleration requires only a step change in force. Vehicle dynamics dictate that an operational controller is more likely to achieve the latter. The choice of control variables is not crucial to the model. However, we expect that it might be difficult in some robot implementations to provide frequent command updates at the tactical level, so acceleration commands would prove to be more convenient.

Lane Tracking. The operational level is assumed to be able to drive the robot down a lane without intervention from the tactical level. This function involves only steering, as do lane changing and intersection traversal. As described earlier, the tactical level adjusts the speed independently to keep lateral accelerations within limits on turns.

Lane Changing. Lane changing requires the operational level to detect the adjacent lane and begin a double-curve maneuver to move the robot over. When the robot reaches the new lane, lane tracking automatically takes over. At most speeds, the steering angle is adjusted to limit the robot's lateral acceleration to a safe value. This constraint results in a lane change maneuver that takes a roughly constant amount of time, independent of speed. The tactical level can use this fact to estimate the length of a lane change. However, if the robot is changing speeds (e.g., braking while approaching an intersection), it is impossible to predict exactly how far the robot will travel during the maneuver. At low speeds, robot can only increase the steering angle to the physical stops of the vehicle. This angle determines the minimum lane change distance for a given road width.

Intersection Traversal. Since intersections generally do not have marked lanes, the

operational system is required to find a path to the given departure lane and drive the robot along the path. As with the other operational functions, there are several ways this may be accomplished. We expect that the robot would follow an imaginary path created from general knowledge about the size of the intersection, and then start up lane tracking when the new lane became clearly visible.

4.1.2. A Future System

The driving system illustrated in Figure 4-1 is intended to be a simplified for studying tactical driving. A more realistic autonomous driving agent would need more functionality, richer communication between components, and a new type of reasoning layer to detect problems.

Functionality. A complete driving system would need more functionality, especially at the strategic level. The nominal strategic component described above merely feeds a fixed route to the tactical level. In a better system, the strategic planner would be able to create flexible routes that were conditional on landmarks rather than using fixed turns at each intersection. The route could be updated dynamically as landmarks or special situations were encountered. This strategic planner would be able to take common instructions such as "follow this road until the gas station, then turn right" and make a tentative plan for the tactical level. When the gas station was found, the route would be updated on the fly.

Communication. A fully functional driving system would also require direct communication from lower levels up to higher levels. In particular, to accomodate real-world malfunctions and errors, lower levels would probably have to report problems to the higher levels. It would be possible for higher levels to detect failure conditions at lower levels indirectly, but it would be easier for the level at which the failure occurred to determine *why* the failure occurred. For example, deceleration failure due to slippery road conditions detected at the operational level should be reported to the tactical level so that braking parameters could be adjusted. Also, a sudden swerve to avoid an obstacle might have to be reported if it results in an unexpected lane change that would otherwise confuse the tactical level. Similarly, the tactical level might be unsuccessful at making a turn at an intersection because of an unexpected road closure or heavy traffic; this should be reported to the strategic level so that it could change the route plan or mission. In fact, there might also have to be communication between non-adjacent levels. For example, excess wheel slippage detected at the operational level might warrant a change of route at the strategic level so that bigger or flatter roads are used.

Error monitors. The error detection capability described above requires extensions to each level of the driving system. The extension to each level reasons about the success of

activities at that level. As such the extensions constitute another layer of the driving system, but along a separate dimension from the strategic-operational dimension. At the operational level, the error detector would look at the set points and expected behavior and monitor the sensed data. If conditions strayed to far from the expected results, the nature of the error would be reported to the tactical level. The tactical level might then reset appropriate physical parameters. Similarly, the tactical level would need a problem detector to determine when the robot was stuck and could not complete the next route instruction.

4.2. Execution Control

Sensing and reasoning activities take time to perform on any real robot. While this is happening, the dynamic world continues to change. Robot behavior is thus dependent to some degree on how its deliberation time interacts with changes in the world. Every robot system design should take this into account. This section explains what assumptions were made for the implementations of Ulysses.

For this research Ulysses was implemented as a simple "sense-plan-act" system. That is, the robot works in cycles. At the beginning of the cycle, the robot senses the world; next, it chooses an action; and finally, it executes the action. The cycle period is governed by a clock, rather than the completion of some action, and takes 100 milliseconds. I have assumed that the system has ideal components that can always observe everything around the robot and finish deliberating within the 100 milliseconds. Such an ideal system can respond perfectly to changing situations in this domain—both because it will notice anything that changes and will determine the best possible reaction (that it knows). At the same time, this system is very simple and allows me to concentrate on the issues of selective perception.

The big question about such an ideal system is how it can actually do all of the necessary deliberation and sensing so fast. AI planning research has generally been concerned with the deliberation part of this question—how to finish deliberating in a short or bounded time (e.g. [Kaelbling 88]), how to produce better plans over time (e.g. [Dean 88]), how to reason about resource usage (e.g. [Horvitz 89]), or how to speed up performance over time (e.g. [Blythe 89]). Robot researchers, on the other hand, have often found that perceptual processing requires far more computation than "planning" on real mobile robots [Shafer 86]. In this research I have found that for tactical driving, the reasoning process alone—as performed by the PHAROS zombies—can easily be performed in real time. Therefore in this research I assume that tactical driving decisions *can* be made in a short cycle period. Thus the driving program has no need to reason about its own planning time, etc. Perceptual processing, on the other hand, is potentially the slow, limiting, and intractable part of the system. The Ulysses implementations described here assume that the robot has whatever resources it needs in order to perceive what the reasoning system requires. The point of this

research is to reduce the resources required for perception in each decision cycle. The following chapters will introduce methods for reducing perceptual computations and estimate their effectiveness by comparing the computational cost to that for general, unguided perception as presented in Chapter 1.

4.3. Perception Control

In many planner based robot systems, planning and perception are independent modules of the agent [Crowley 85, Fikes 72, Firby 87, Georgeff 87, Laird 89, Lin 89, Nitao 86, Schoppers 87]. Figure 4-2 shows basically how the modules are related. In these systems the planning component works on a symbolic world model, which it assumes the perception component keeps up to date. There is no communication from the planner to perception. The perception component must therefore find *everything* of potential interest to the planner and dump it into the world model. In a dynamic world—and especially, with the robot moving—the apparent state of the world potentially changes all the time, so perception must update the world model continuously (every decision cycle). This is the system architecture assumed in Chapter 1 when the cost of general perception was estimated.

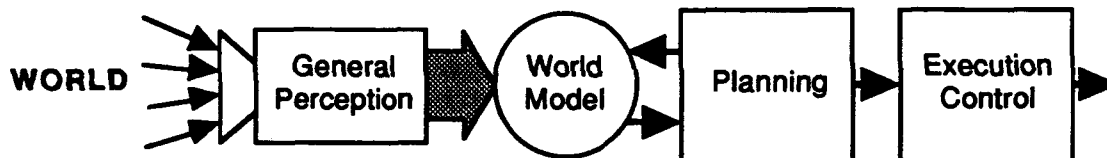


Figure 4-2: A traditional robot control system.

I have demonstrated previously how impossibly expensive perception is in this traditional system design. There are three significant shortcomings in this type of system which prevent perception from being more efficient:

1. There is no way for the reasoning module to communicate with the perception component to command sensing actions, or ask for specific information and give hints about where to look.
2. The perception module may provide much redundant information beyond what is needed to determine a unique robot action.
3. The reasoning system does not take advantage of coherence in the world over time to retain information in the world model for later decision cycles. Thus some sensed information may be redundant with data sensed earlier.

The next three chapters describe implementations of Ulysses that address each of these shortcomings. The implementations are called Ulysses-1, Ulysses-2, and Ulysses-3, respectively. Ulysses-1 introduces a language for requesting information about specific types of objects. The domain-specific requests implicitly describe where to find objects in the scene.

Ulysses-2 intelligently selects which sensing requests to make. Ulysses-2 looks for objects that are important for the task in the given situation. The perception system is not asked to look for other objects. Ulysses-3 marks all sensed information with a time stamp and remembers it. Domain-specific knowledge about the dynamics of objects is then used with the inference tree to determine when objects must be re-sensed. The resulting architecture is shown in Figure 4-3.

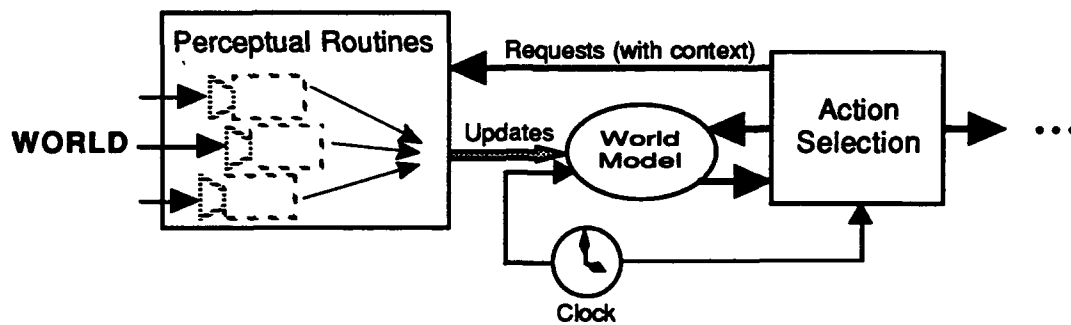


Figure 4-3: A control system with selective perception.

Other robot researchers have recognized the need to make sensor control part of the reasoning process, but this work has generally been limited to very simple domains. Firby's work [Firby 89] is very similar in spirit to mine. He proposes to use task-directed sensing both to limit information processing requirements and to reduce uncertainty. His RAP system (for "Reactive Action Package") can generate sensing requests during execution. The requests can be for specific objects at designated locations. In addition, facts in the world model have time stamps on them so the system can determine how old they are. However, sensing control in this system is intended more to test a few predicates—e.g., the preconditions of a manipulation operator—rather than make complex assessments before each action, as Ulysses does. The RAP system would require that any complex sensing strategies be programmed in advance, instead of falling out of an inference process during execution (see description of Ulysses-2 in Chapter 6). Robot planners that insert sensing actions into existing plans also use simple sensing actions to verify conditions [Brooks 82, Doyle 86]. These planners are even less flexible than Firby's in that they cannot choose manipulation or sensing actions interactively during execution.

Several real mobile robot systems have used perception control during execution, but have not explored the problem of selecting critical objects. For example, the Martin Marietta ALV [Lowrie 85] and Carnegie Mellon University NAVLAB [Shafer 86] both had perception components that accepted requests from a reasoning component. These requests could specify object types and geometric areas in which to search. However, these systems mostly just followed roads and did not have the need to look at many objects.

Some researchers have worked on the problem of limiting sensing actions in complex tasks. However, to date this work has only been demonstrated in simple environments. For example, Chrisman and Simmons [Chrisman 91] describe a method for choosing cost-effective, selective sensing strategies; however, it is not clear whether their decision theoretic technique could be applied in a complex domain with many states. Hayes-Roth's BB* system [Hayes-Roth 90] reasons about computational resource limitations and adjusts *filters* on the data input stream. These filters can limit the sampling rate to reduce data flow. This technique implicitly assumes that the sensors can detect everything by default—an invalid assumption in a complex environment such as driving.

Although machine vision research has addressed various uses of sensor control [Aloimonos 88, Ballard 91], this work has emphasized foveated image processing, gaze control, and eye-hand coordination instead of task-based visual search control. For example, several mobile robot vision systems [Akatsuka 87, Griswold 89, Kluge 89, Solder 90] use small windows so that they have less data to process when looking for signs or other objects. These systems demonstrate only minimal intelligent perception control though, since the tasks required that the robot always look for the same object—for example, monitoring a lane line or watching for a speed limit sign. Burt [Burt 88] also advocates using small windows, but with the ability to move around the image to probe interesting areas. However, he offers no specific algorithms for generating this control from a given task; his emphasis is on lower-level techniques such as data compression, reduced image resolution, hierarchical analysis, etc. Finally, there has been work in human vision that addresses perception control as it relates to cognitive tasks [Yarbus 67]. This work provides interesting descriptions, but does not offer a computational model for generating behavior in an autonomous robot.

4.4. Summary

Ulysses is a model of tactical driving that is intended to be part of a complete driving system. In this research, the strategic and operational levels are abstract. The strategic level provides a detailed route plan for Ulysses, and the operational level flawlessly implements a simple set of movement and tracking commands. Independent perception resources are available to each level.

The Ulysses implementations use a simple "sense-plan-act" cycle to control robot activity. That is, at each clock tick Ulysses can perceive and assess the situation from scratch in order to choose the best possible action. Since the Ulysses model determines actions from the current situation without considering hypothetical future worlds (i.e., planning), and since uncontrolled perception is so expensive, I assume that perception will be the resource bottleneck instead of reasoning. Thus the driving robot will need to actively control perception to limit its computational demands.

This thesis shows how a robot can use task knowledge to control perception. Perception control is demonstrated in three stages. Ulysses-1 introduces a language of task-specific perceptual requests that implicitly use spatial relations to limit search and find the important objects. Ulysses-2 reasons about an inference tree to determine which objects are important and which can be ignored by perception. Ulysses-3 uses knowledge about domain dynamics to determine when objects must be sensed again. These perception control techniques are new in that they incorporate task-level knowledge instead of just low-level gaze control, data reduction and data compression concepts. The Ulysses implementations are described in the next three chapters.

Chapter 5

Ulysses 1: Perceptual Routines

This chapter describes Ulysses-1, the first implementation of Ulysses. Ulysses-1 addresses the fact that a classical planner does not have any way to direct perceptual actions. Ulysses-1 incorporates a mechanism for the reasoning module to request specific perceptual actions to sense important objects. These requests are defined in a task-specific way to allow the perception module to limit its search of the visual field. This chapter introduces the concept of perceptual routines, discusses how Ulysses-1 uses them to implement the driving model, and describes how routines reduce perceptual cost in various driving situations.

5.1. Defining the Perceptual Language

5.1.1. Routines

Section 3.4.1 described how tactical driving requires information about spatial relations between objects. For example, the reasoning system might want to know if there is a car approaching a downstream intersection from the right. There are at least two ways to find the objects in the desired relations. One way would be to detect all cars, and then test each car to see if it was on the road to the right at the intersection. "The road to the right" would itself be found by finding all roads and checking to see if they connected to the the robot's road at the intersection in question. This method would be very difficult because there may be many cars and roads in the robot's field of view, and each one requires significant computation to find and test.

Ulysses-1 (and the later implementations of Ulysses as well) uses a different technique to detect specific objects in specific relations to one another. The perception subsystem uses the reference object and a relation to focus its search for a new object. In the example above the robot would *track* the corridor ahead to the second intersection, *scan to the right* to find the approach road, and then *look along the road* for a car. The car detection process is thus limited to a specific portion of the field of view, determined by the location of the road. Ulysses-1 uses about a dozen of these routines, as shown in Table 5-1. Although it may seem that driving decisions could be made with simpler, more general sensing functions, Figure

Find current lane	Find next car in lane
Mark adjacent lane	Find crossing cars
Track lane	Find next sign
Profile road	Find next overhead sign
Find intersection roads	Find back-facing signs
Find path in intersection	Find signal
Find next lane marking	Marker distance

Table 5-1: Perceptual routines in Ulysses-1.

3-23 shows that very simple sensing is not sufficient; for example, a low-level operator to "detect converging objects to the right" could not distinguish between the two intersections.

5.1.2. Related Work

Ullman describes how routines may be used in the human vision system to determine object properties and spatial relations [Ullman 84]. These visual⁴ routines are composed of operations such as shifting processing focus, finding unique locations in the image, tracing the boundary of a contour, filling a region to a boundary, and marking points for later reference. While the input to such routines is an image processed from the bottom-up, the routines themselves are invoked from the top-down when needed in different tasks. For example, the routines can be used to determine if two points lie within the same contour. Agre and Chapman used visual routines in their video game-playing system, Pengi [Agre 87]. These routines used the primitive actions described by Ullman. The Pengi planner executed visual routines just like any other actions in order to get information about the world state. Since the planner used rules that always related objects to one another or the agent, the visual routines were able to find important objects directly by computing the appropriate spatial relationships. For example, a visual routine could find "the block that the block I just kicked will collide with."

The perception system for Ulysses-1 was inspired by Pengi. As we described above, the

⁴We use the term "perceptual routines" instead of "visual routines" in our work to emphasize that they may include higher levels than just low-level vision, and that they may include other sensing technologies.

rules for driving also have spatial relations incorporated in them. Ulysses-1 uses routines to directly find roads related to the robot, cars related to the roads, signals related to particular intersections, etc. The routines in Ulysses-1 assume more domain-dependent processes are available to the routines than is the case in Pengi; for example, the routines must understand how to trace roads instead of just contours, and how to stop scanning when a sign is recognized. Most of the routines mark objects and locations when they finish, so Ulysses-1 can continue finding objects later. For example, Track lane stops when an intersection is encountered (indicated by the change in lane lines in the well-marked PHAROS world), but marks the intersection so that Find path in intersection, Find signal, etc. can find objects relative to that intersection. The next section provides a detailed example of how Ulysses-1 uses routines.

Agre and Chapman point out two advantages of routines in terms of knowledge representation. First, only relevant objects in the world need to be represented explicitly in the agent's internal model. Pengi, for example, does not have to model all of the blocks on the video screen. Second, when the planner needs to know if there is an object with a certain relation to a reference object, it does not need to check all visible objects to find out.

These representational advantages have perceptual analogs which are illustrated better by Ulysses-1 because it addresses perception more realistically. First, not only do perceptual routines avoid having to represent all world objects internally, they avoid having to *look* for all the objects. In Pengi perception was abstract and essentially cost-free; for a real driving robot, looking for things is extremely expensive and any reductions in sensing requirements are important. Second, checking the relationship between two objects is more complicated than, say, looking at a value in a property list. Computing arbitrary geometric relations may be difficult, and it is much better to do it once and find the appropriate object directly. For example, finding a car on a road could involve making a region-containment test on every visible car; it would be better to scan the road region until the car was found.

The routines listed in Table 5-1 are clearly specialized for the driving task. They are designed to allow the domain-specific knowledge in the planner to be used for controlling perception. A system that uses such routines would use different routines for different tasks. This perception philosophy is similar to the idea of task-oriented vision described by Ikeuchi and Hebert [Ikeuchi 90].

5.1.3. How Routines Are Used

In the Ulysses driving model the role that traffic objects play depends on their relation to the corridor ahead of the robot. The corridor is the series of lanes and intersections downstream of the robot that follows the robot's intended route. Ulysses-1 locates the

corridor at the beginning of every decision cycle. This process is very straightforward, involving the repeated application of road-finding routines until the range limit of the sensors is reached. For each part of the corridor, Ulysses-1 applies all active constraints and preferences. These are encoded in individual subroutines. Whenever these subroutines refer to an object in the world, Ulysses-1 requests that a perceptual routine be run to look for the object. Since Ulysses-1 is not using any knowledge to model changes in the world, all constraints and preferences are active almost all of the time⁵. Thus the robot ends up requesting information about all objects of potential interest along the corridor.

The remainder of this section illustrates the use of routines with an example. Figure 5-1 shows a driving situation in which the robot is approaching a left-side road and must turn left. Traffic can appear on any road. There are a few speed limit signs, route signs (not shown in the figure), a STOP sign, a "left turn only" sign overhead and a "left turn only" arrow on the pavement. The robot's road has four lanes, while the side road has only two.

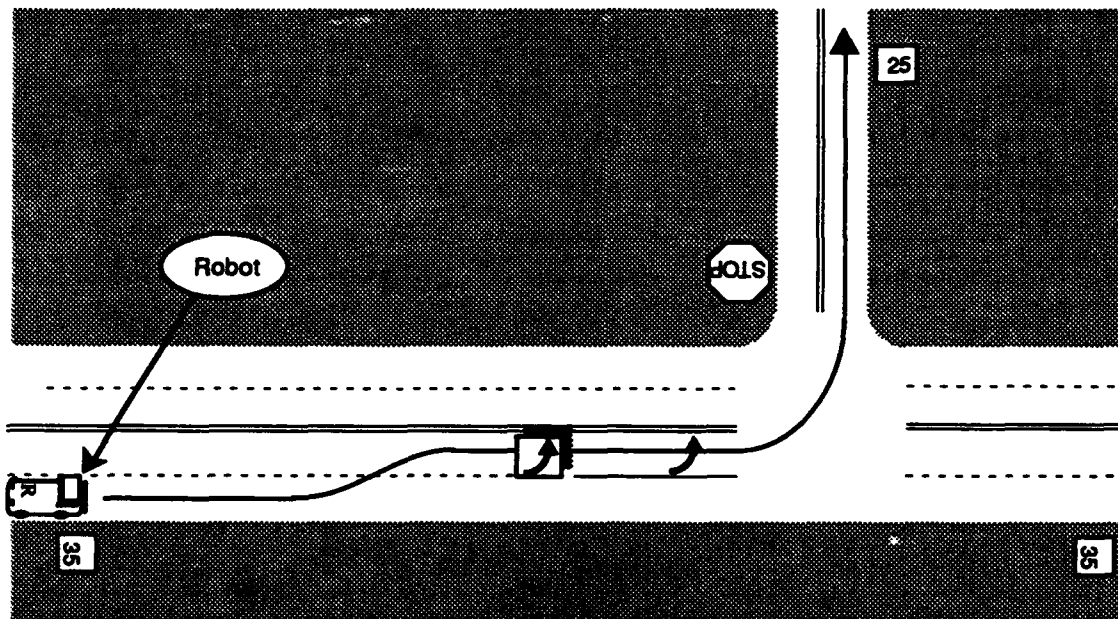


Figure 5-1: Left side road scenario (Not to scale).

The following series of figures shows the perceptual activity during *one decision cycle* of the robot. We pick up the robot after it has changed lanes and is approaching the intersection (Figure 5-2). There is a car approaching from the through direction, one in the adjacent lane behind, and one in the oncoming lanes (having turned from the side road).

⁵There are some exceptions: Ulysses does not consider lane-changing actions when the robot is already in the process of changing lanes, or if there is only one lane.

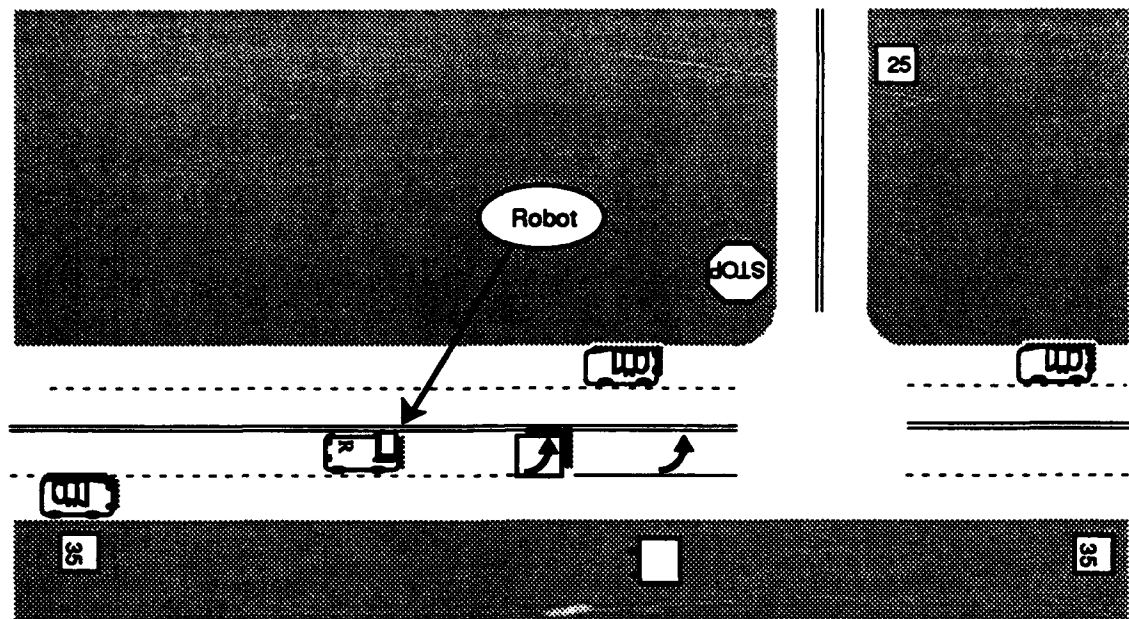


Figure 5-2: Partway through the left side road scenario.

The first thing the robot does is find the lane immediately in front of it using the `Find current lane` routine. It then uses the `Track lane` routine to look forward to obtain the first section of the corridor. The scan of the lane ends at the intersection. The robot then looks for cars in the lane with `Find next car in lane` and signs along the side of the road using `Find next sign` repeatedly. Figure 5-3 shows the areas scanned by these four routines.

Figure 5-4 depicts the process of determining the channelization of the robot's lane (i.e., whether it can turn left from the lane). Ulysses-1 scans the width of the road—using `Profile road`—at the intersection (at the marker left earlier) to determine the position of the robot's lane at the intersection. The robot also looks for signs over the lane and markings in the lane to indicate turn restrictions (`Find next overhead sign`, `Find next lane marking`). If such objects are found, these scans are repeated until the end of the lane is reached. Markers indicate where one leaves off (at an observed sign or marking) and the next one starts.

In Figure 5-5 Ulysses-1 is finding the next part of the corridor and analyzing the intersection itself. This requires finding a path in the intersection (routine `Find path in intersection`), looking for lead cars on this path (`Find next car in lane`), looking for crossing or obstructing cars (`Find crossing cars`), looking for traffic signals around the intersection (`Find signal`), and identifying all approach roads (`Find intersection`

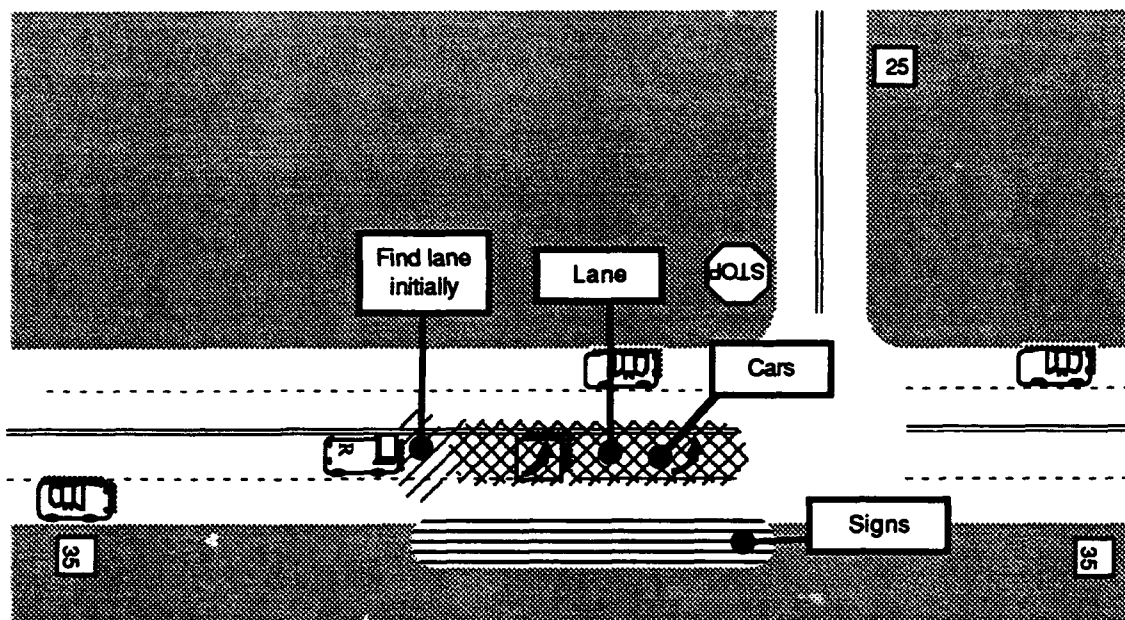


Figure 5-3: The robot scans for the lane, cars, and signs ahead.

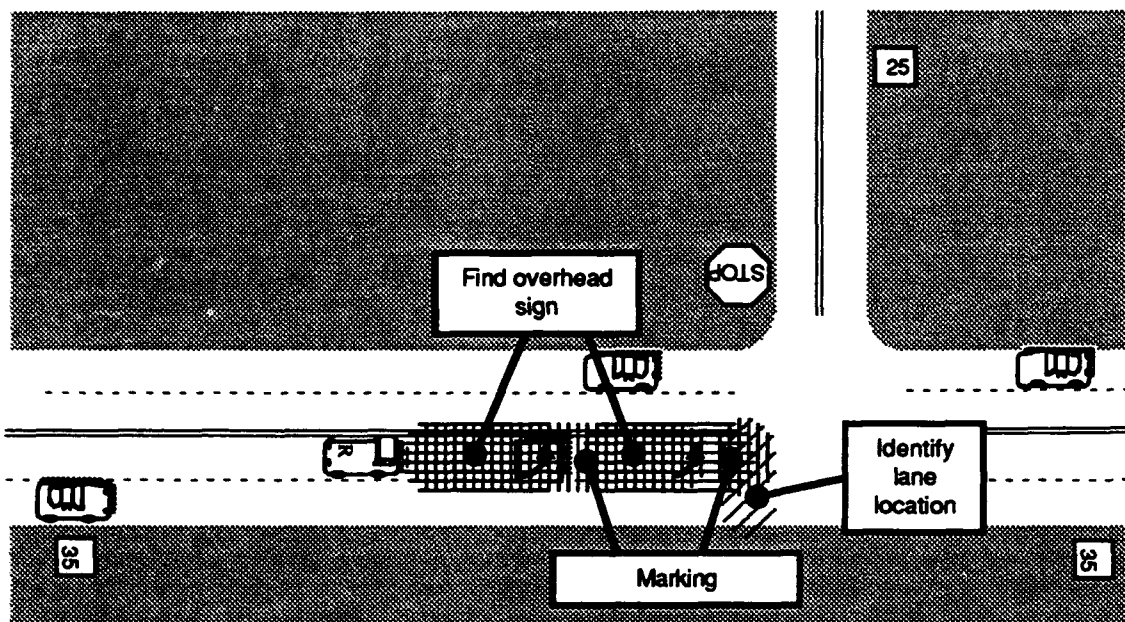


Figure 5-4: The robot checks lane position and looks for channelizing signs and marks.

roads). Information Ulysses-1 already has allows it to determine whether there is a traffic

control sign facing the robot, and how many lanes the robot's road has. Lane counts help determine which road is bigger and are part of the right-of-way determination process.

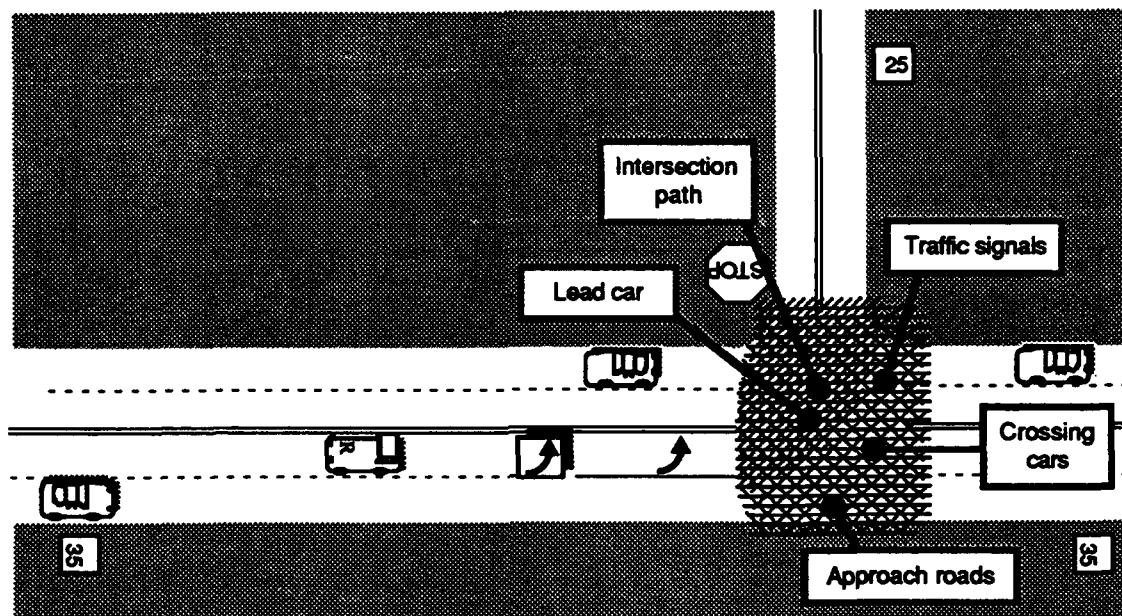


Figure 5-5: The robot looks for a path in the intersection, and then cars, signals, and approach roads.

Figure 5-6 shows areas scanned to check for approaching traffic on each road, and for STOP or YIELD signs facing this traffic. Ulysses-1 first finds the lanes on the approach roads, and then looks for the closest car to the intersection in each lane. Approaching cars are found using Find next car in lane again, and the signs are detected with the Find back-facing signs routine. The distance between the approaching cars and the intersection can be determined with Marker distance.

In Figure 5-7 the robot is looking beyond the intersection to its downstream road. As with the first piece of corridor, Ulysses-1 first finds the lane, then looks for cars and signs. In this case one sign is found.

At this point Ulysses-1 has examined everything necessary to constrain the robot's speed. The remaining steps are performed to select a lane. Ulysses-1 considers lane selection even as the robot approaches the intersection because in some situations it may be desirable or necessary to change lanes to move around a slow car. Figure 5-8 shows that Ulysses-1 finds the adjacent lane (to the right only, in this case) and analyzes it much as the robot's lane. The adjacent lane is found and marked using Profile road and Mark adjacent lane. Ulysses-1 tracks the lane ahead to the intersection, and looks for cars, overhead signs (signs

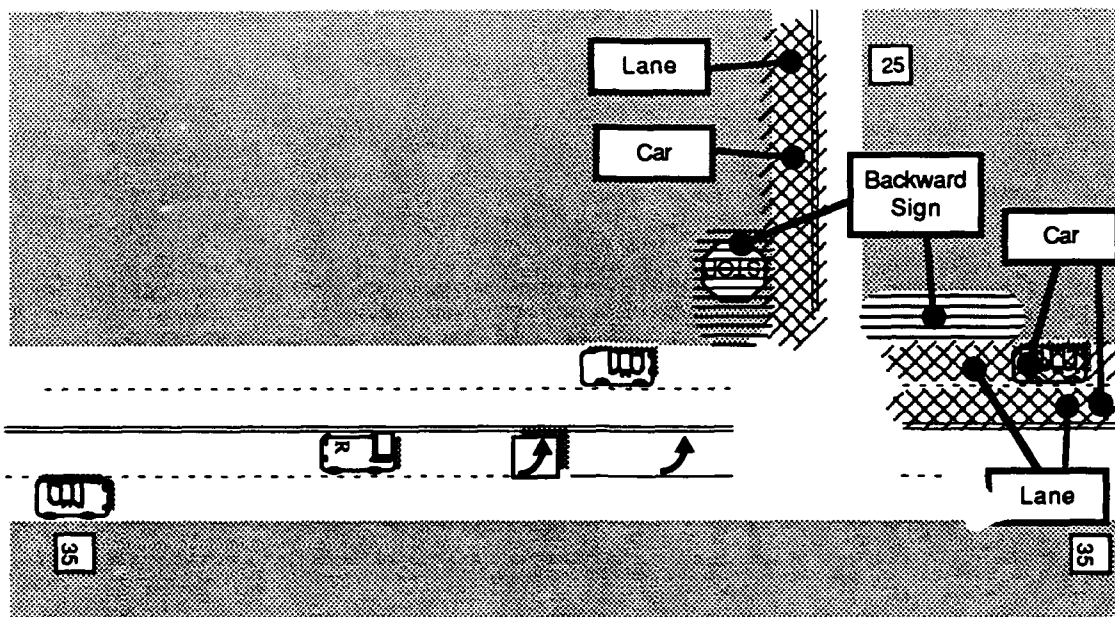


Figure 5-6: The robot checks the intersection approaches for traffic and signs.

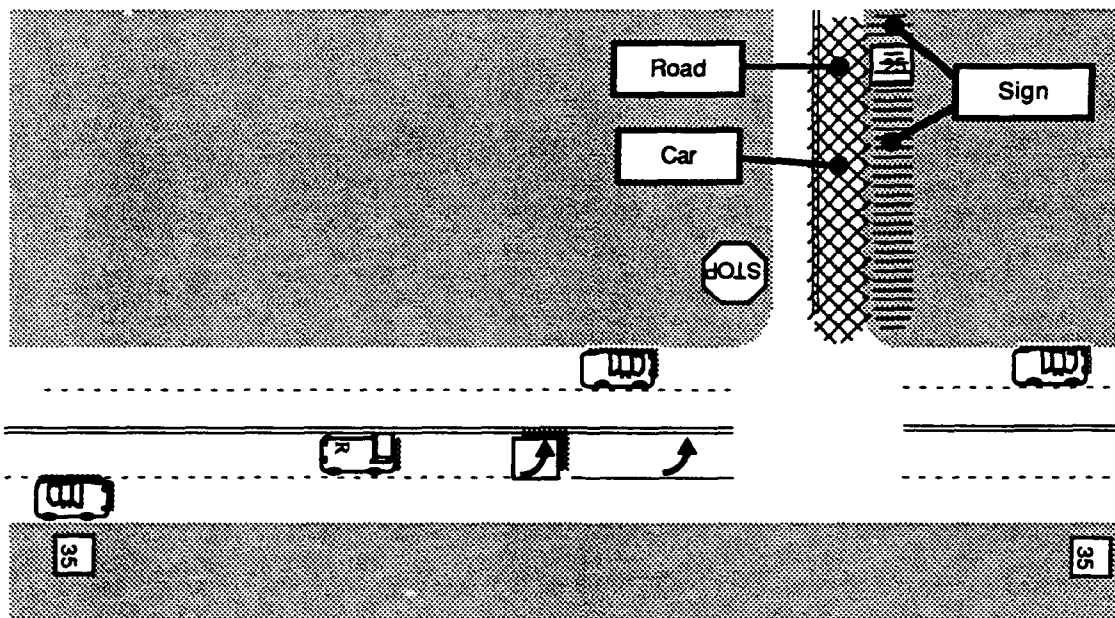


Figure 5-7: The robot looks for a lane, cars, and signs downstream of the intersection.

to the right have already been found), markings, and lane position at the intersection. The robot also looks in the lane behind to find the car behind.

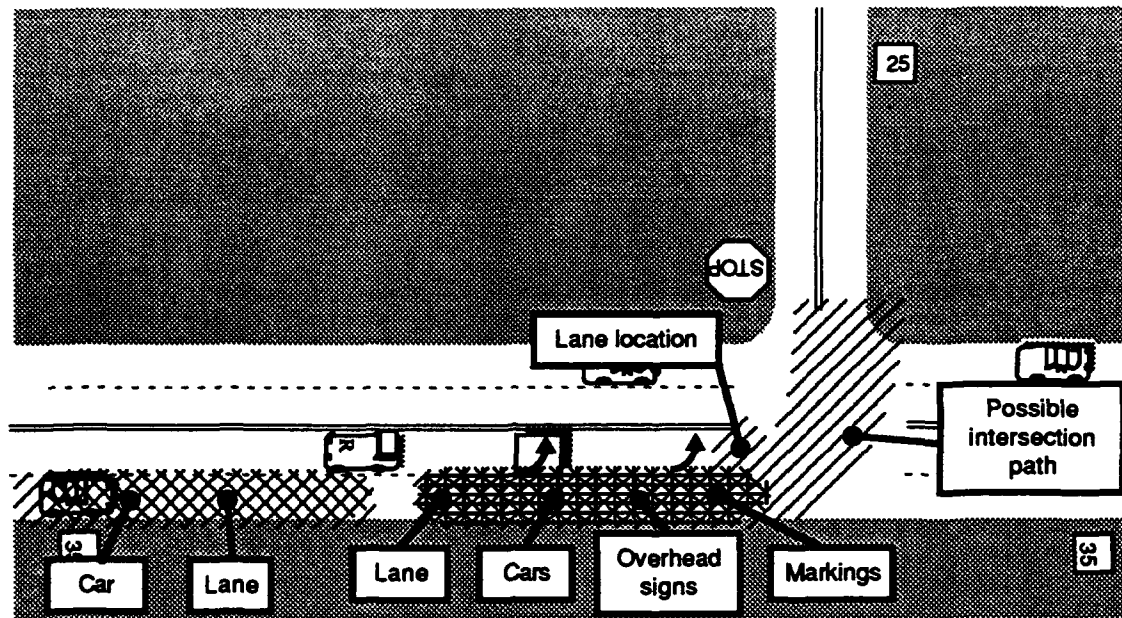


Figure 5-8: The robot looks forward and backward for constraints affecting the adjacent lane, and traffic in that lane.

Ulysses-1 performs the visual search described above in order to find all objects that could potentially constrain the robot's actions. While the search seems exhaustive, it is nevertheless restricted to specific areas around the corridor. Because Ulysses-1 has no knowledge about domain dynamics, it must repeat this search every clock tick to guarantee that the robot can respond adequately to changing situations. Thus Ulysses-1 performs this search *every 100 milliseconds*.

5.2. The Cost of Using Routines

5.2.1. General Cost

The perceptual routines are subject to the same environmental conditions as a general, naive perception system, so must pay the same feature extraction costs. However, the use of routines can reduce perception costs in several ways:

- **Reduced search area.** The azimuth and elevation angles swept by the routine are much smaller than the area searched by the naive perception system.
- **Range-limited resolution.** The maximum depth reached in the routine's search area limits the resolution required.
- **Object-limited resolution.** The routines look for only one type of object at a time; resolution is determined by the size of that type, not by the smallest of all types.

- **Limited features.** Since the routines look for only one type of object at a time, they may be able to use a small set of features that are specific to that object type. Only these features need to be extracted from the image.

The effectiveness of these reductions depends on the situation; for example, if the robot used several routines to search the same area for different objects, it would not get the benefit of limited features or object-limited resolution.

The cost of using routines can be estimated using the same general assumptions that were used for a general perception system in Chapter 1. Although in some cases the more specific routines could use simpler recognition algorithms (e.g., a "car" is any blob on the road), I have assumed that they use the more general algorithms. The cost of individual routines depends on the feature size of the objects they detect (from Table A-1), and the specific area searched by the routine (as illustrated by Figures 5-3 through 5-8 above). Since the cost depends on the search area, there is no fixed cost for most routines. Appendix B explains in detail how to calculate the cost for each routine. The following section illustrates how the cost of the routines changes over time as the robot's situation changes. Appendix C shows the cost of the individual perceptual actions during the single decision cycle described in the previous section.

5.2.2. Driving Experiments

I have created several driving situations in PHAROS and used Ulysses-1 to drive a robot through them. By tracking the perceptual costs over time, we can see how the cost of routines compares to the cost of naive perception, and what situations cost the most with the given routines. PHAROS simulates the operation of perceptual routines and estimates the cost of each routine when it is called. The scenarios are as follows:

- The left-side-road scenario described above. This scenario includes a lane change before the intersection and the detection of a conflicting car with right of way. It requires a search for traffic control devices at the intersection.
- A four-way intersection with no traffic control (unordered intersection). Illustrates visual search for different right-of-way logic.
- A four-lane highway with no intersections. This scenario removes the complication of intersections, but includes car following and passing actions.
- An intersection of four-lane roads controlled by traffic lights. This scenario includes many signs and markings and more cars than the other scenarios. It also illustrates an intersection with completely different traffic control.
- A set of closely spaced intersections of two-lane roads. This scenario includes Stop and Yield signs for the robot, and requires consideration of two downstream intersections at once. There is also an intersection on an approach road to complicate the search for conflicting cars.

All of the scenarios contain signs, markings, etc. that the robot can observe. In addition, there are times in these scenarios during which the robot is driving on two- or four-lane roads and cannot see an intersection.

5.2.2.1. Left Side Road

Figure 5-9 illustrates the left side road scenario introduced earlier; Figure 5-10 is the graph of the estimated perceptual cost over time. The circled numbers in road picture are reference points that correspond to the labeled points in the graph. These values were computed as described in Appendix B by the interface between PHAROS and Ulysses. The cost of perception is dominated by the lane search cost, which in the figure is indistinguishable from the "Total cost" plot. This relation is also true in all other scenarios. Searching the lane is the most costly type of activity because foreshortened lane and stop lines are the smallest objects in the environment (see Appendix A). Since this version of Ulysses-1 does not use any knowledge about event horizons, but naively searches to its sensor range limits, the cost of finding lanes can be very high.

The perceptual cost before point (1) in Figure 5-10 is approximately 1.5×10^{17} operations per second; this represents the cost of a four-lane highway without intersections. At point (1), the sensors detect the intersection and trigger additional searches for signals, approach roads, and blocking cars. The figure shows that the signal and car search cost jumps up here. Ulysses-1 also determines that the robot cannot turn left from the right lane and begins a lane change. During this time, the robot stops searching the adjacent lane for traffic, so the cost of searching lanes drops sharply. At point (2), the sensors detect the robot's corridor on the far side of the intersection and begin searching for signs and cars on that road. At (3) the sensors can detect all approach roads, so the robot begins to search these roads for lanes, cars, and backward-facing stop and yield signs. After the lane change is complete at (4) the robot again watches both lanes of the road and the lane search cost increases accordingly. This is approximately the point in time described in Section 5.1.3 and in Appendix C. At point (5) the robot enters the intersection and ceases its search for signals, approaching cars, and other objects that would affect its right-of-way decision. At (6) the robot no longer needs to look for unexpected cross traffic in the intersection either.

5.2.2.2. Unordered Intersection

Figure 5-11 shows the unordered intersection scenario. The robot approaches the intersection and must yield to a car on the left before proceeding with its left turn. The car on the right is expected to yield to the robot. There are several miscellaneous signs along the roads.

Figure 5-12 is a graph of the perceptual cost in operations during the scenario. The perceptual cost before point (1) in Figure 5-12 is approximately 2×10^{16} operations per second; this represents the cost of a simple two-lane highway without intersections. After point (1), the robot has detected the intersection and begins to search for signals, roads, cars, and additional signs according to the corridor model. At point (2), the sensors can reach the far

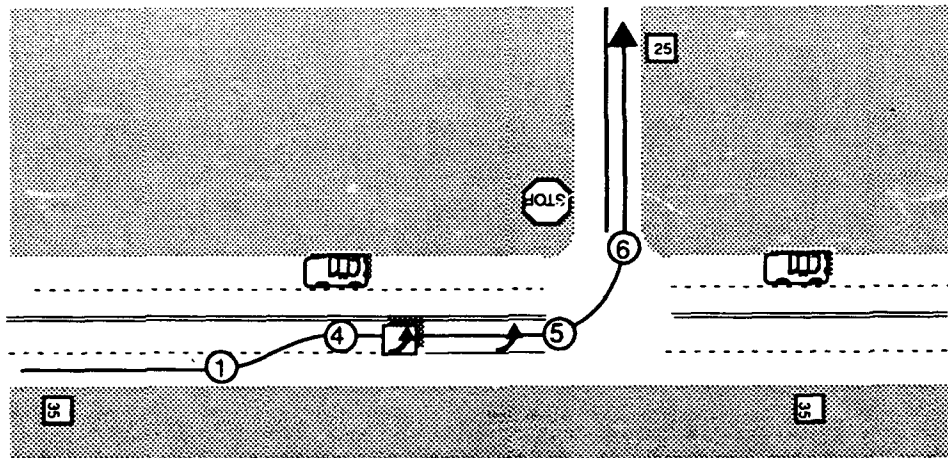


Figure 5-9: Left side road scenario (not to scale).

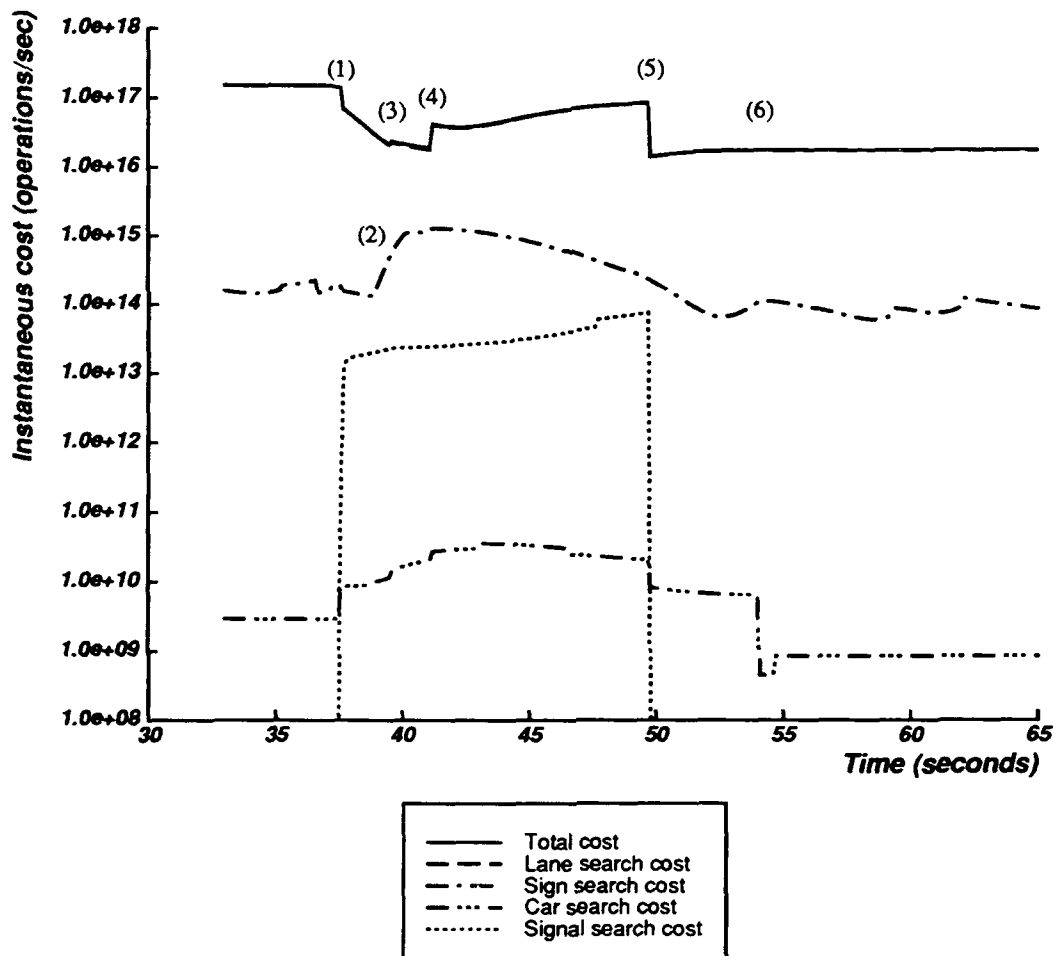


Figure 5-10: Perceptual costs during scenario. Numbers correspond to figure above, except:
 (2) sensors reach robot's destination street on far side of intersection;
 (3) sensors reach all intersection approach roads.
 Total cost is nearly the same as lane search cost.

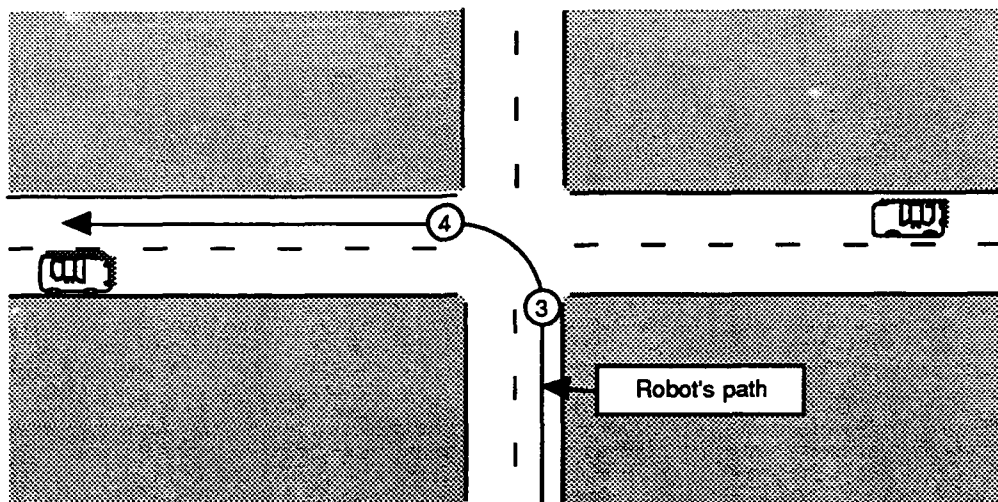


Figure 5-11: Unordered intersection scenario (not to scale).

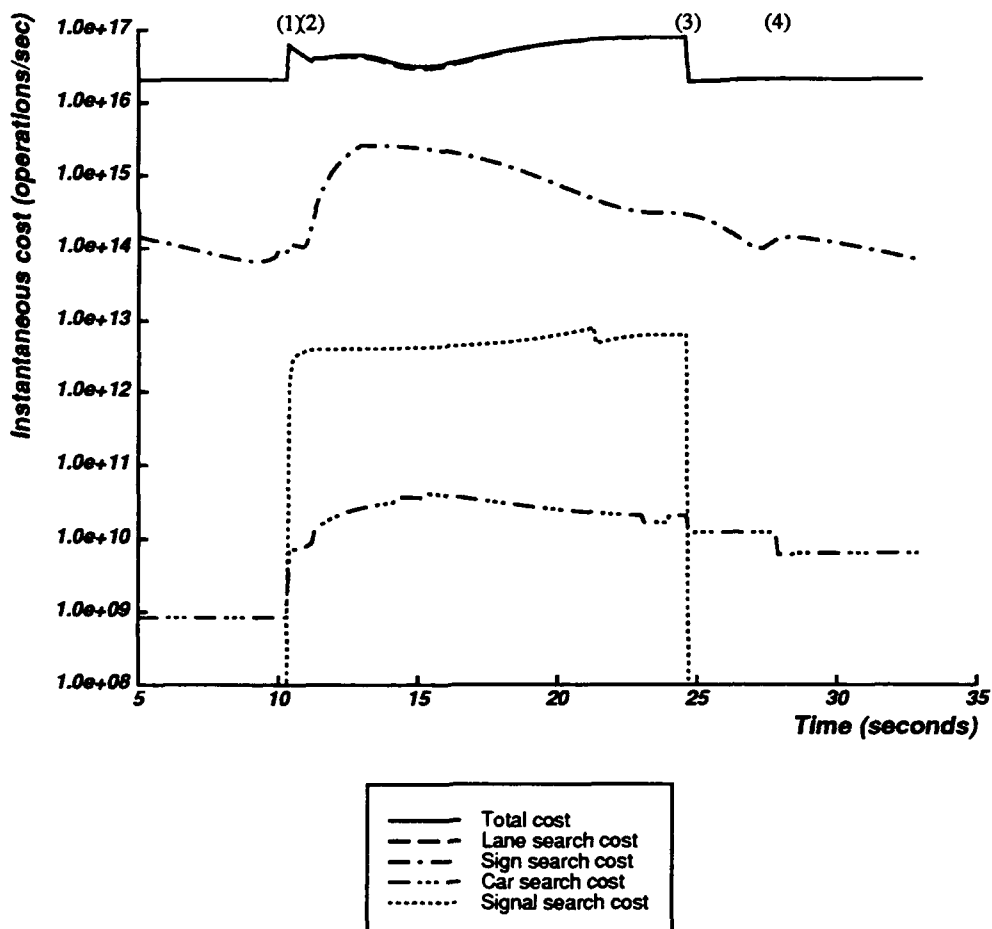


Figure 5-12: Perceptual costs during scenario. Notes correspond to figure above, except: (1) sensors reach intersection; (2) sensors reach all intersection approach roads. Total cost is nearly the same as lane search cost.

side of the intersection and detect the approach roads, so these roads are scanned for lanes, cars, and backward-facing stop and yield signs. At point (3) the robot enters the intersection and ceases its search for signals, approaching cars, and other objects that would affect its right-of-way decision. At (4) the robot no longer needs to look for unexpected cross traffic in the intersection either. After (4), the robot returns to a two-lane highway situation.

5.2.2.3. Four-lane Highway

Figure 5-13 is the four-lane highway scenario. In this case the robot has to pass a slower car using the left lane, and then return to the right lane. There are no other traffic objects except for various signs along the side of the road.

Figure 5-14 shows the estimated costs of using perceptual routines in this situation. The robot detects the car in front at point (1); the car detection cost rises very slightly here because the blob in the road must be identified. At points (2) and (4), the robot commits to a lane change and stops looking behind itself in the adjacent lane for traffic. This causes a sharp decrease in the lane-search and total cost.

5.2.2.4. Intersection With Traffic Lights

Figure 5-15 shows the traffic light scenario. This scenario includes more traffic objects than previous scenarios, including signals and more markings and signs. Not shown in the figure are cars that approach from all directions during the scenario. The robot enters in the right lane and is required to make two lane changes before entering the intersection. The left-most signal head facing the robot shows a left-turn arrow after the robot has waited at point (6) for a few seconds. After the robot crosses the intersection, it again moves to the right-most lane.

Figure 5-16 shows the estimated perceptual costs for this scenario. The graph's features reflect sensor detection events, lane changes, and intersection entries as described for previous scenarios. The costs are not significantly affected by the additional objects in the environment; for example, signal search costs are almost the same in this scenario as in the left side road scenario. The cost of searching the intersection for signal heads apparently dominates the cost of finding lens symbols on the heads that are found.

5.2.2.5. Multiple Intersections

The final scenario includes multiple intersections. Figure 5-17 shows that the robot must traverse two closely spaced intersections, and scan across an intersection on an approach road. At the first intersection the robot has a Yield sign, but determines that the vehicle approaching from the right is stopped and will not conflict with the robot. The robot therefore determines that it can proceed freely through the first intersection. However, the

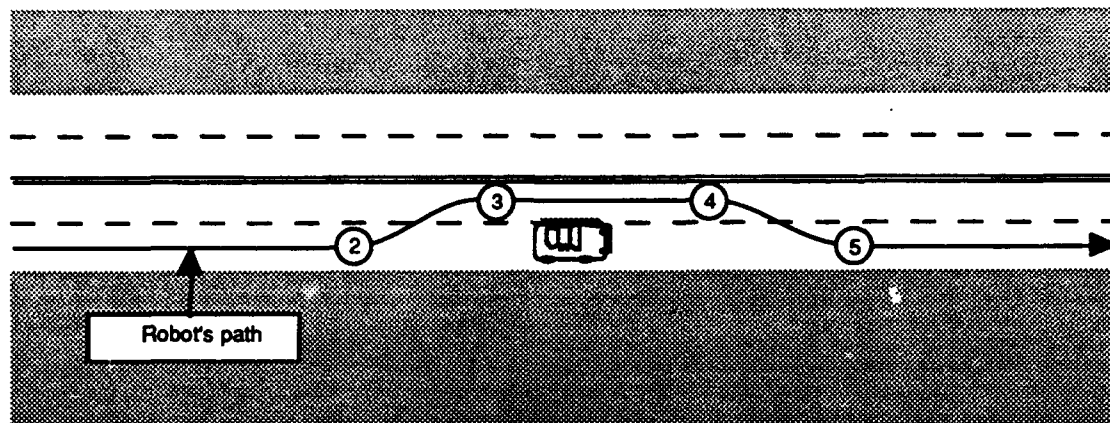


Figure 5-13: 4-lane passing scenario (not to scale).

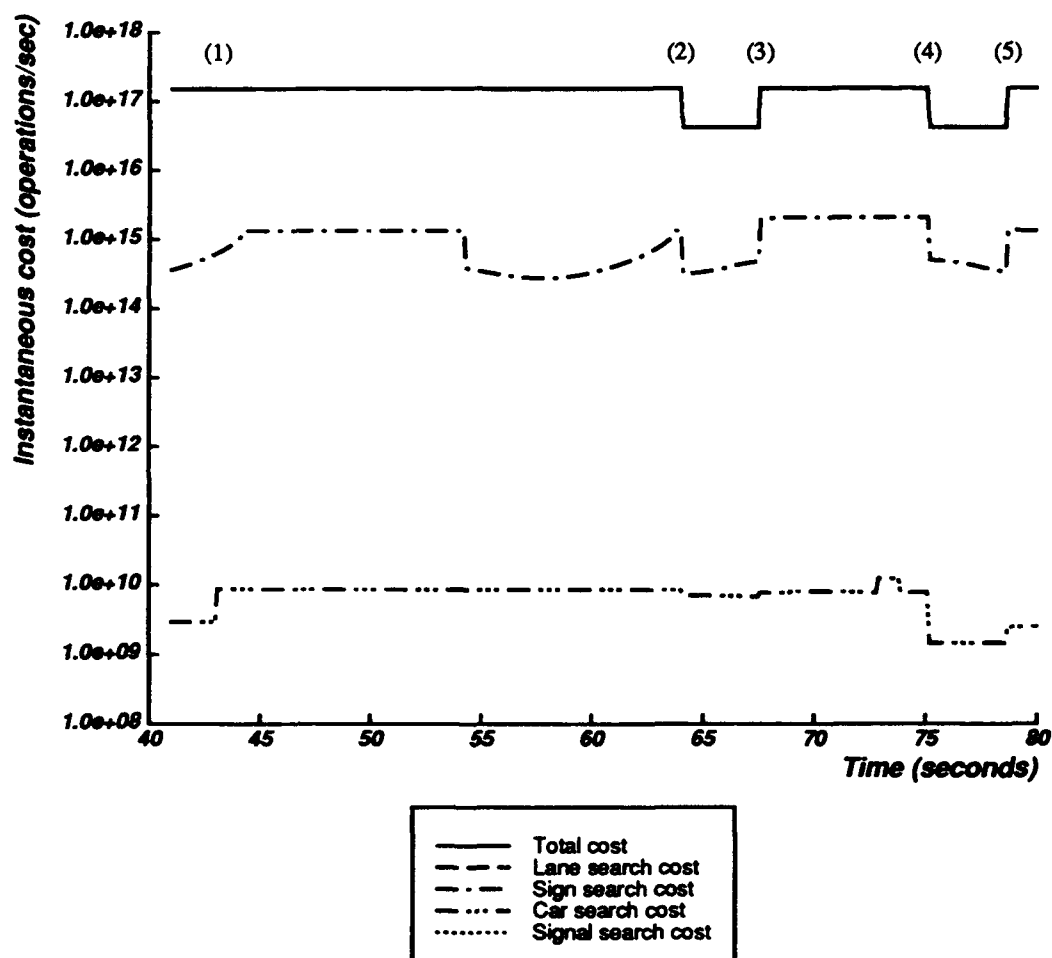


Figure 5-14: Perceptual costs during scenario. Notes correspond to figure above, except: (1) sensors reach lead car. Total cost is nearly the same as lane search cost.

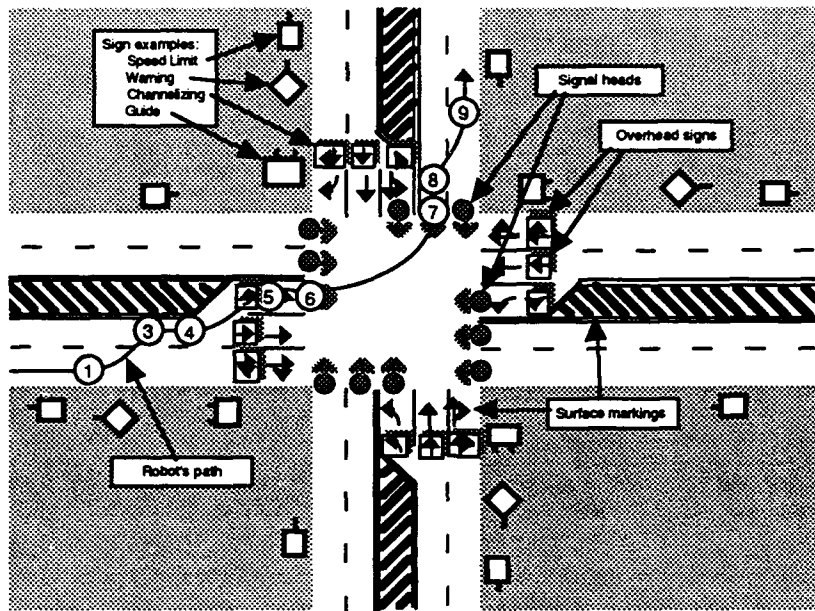


Figure 5-15: Traffic light scenario (not to scale).

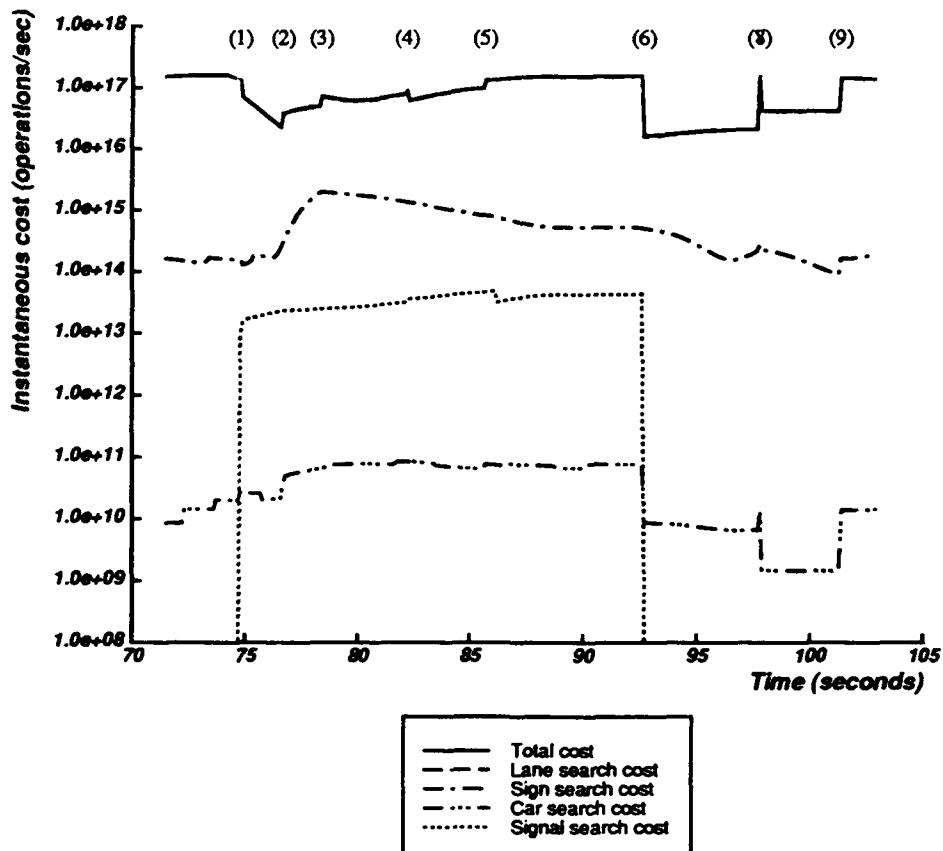


Figure 5-16: Perceptual costs during scenario. Notes correspond to figure above, except: (1) sensors reach intersection; (2) sensors reach all intersection approach roads. Total cost is nearly the same as lane search cost.

Stop sign at the second intersection forces the robot to slow down even before it reaches the first intersection. At the second intersection, the robot waits for crossing traffic to clear before proceeding. Figure 5-18 shows the estimated cost for this scenario.

5.3. Summary

The Ulysses-1 driving system introduces a language for controlling perception. This language is a collection of perceptual routines, each of which performs a specific, limited perceptual action. Routines do not search the entire scene for their target objects, but instead search in specific places relative to other objects. The routines are task-specific, because the reasoning component depends on the semantics of this relative search to get the right objects in the scene. The object relations implicit in the routines are of course designed to match the task. The reasoning component of the system can thus avoid doing the spatial reasoning by pushing it off on the perception component; the routines do the spatial reasoning implicitly in their search of the physical world. The result is a computation savings and a perceptual search savings for the system.

Ulysses-1 attempts to find everything in the world of interest for driving, just like a general perception system. However, it uses perceptual routines to find the corridor and other traffic objects. The routines significantly reduce the amount of perceptual search required to find the relevant objects. Experiments in simulated driving situations indicate that routines require about 10^{17} operations per second, as compared with about 10^{20} operations per second necessary for general perception. Thus even in a relatively naive driving system that continues to sense everything within range every decision cycle, perceptual cost can be reduced by three orders of magnitude by using perceptual routines.

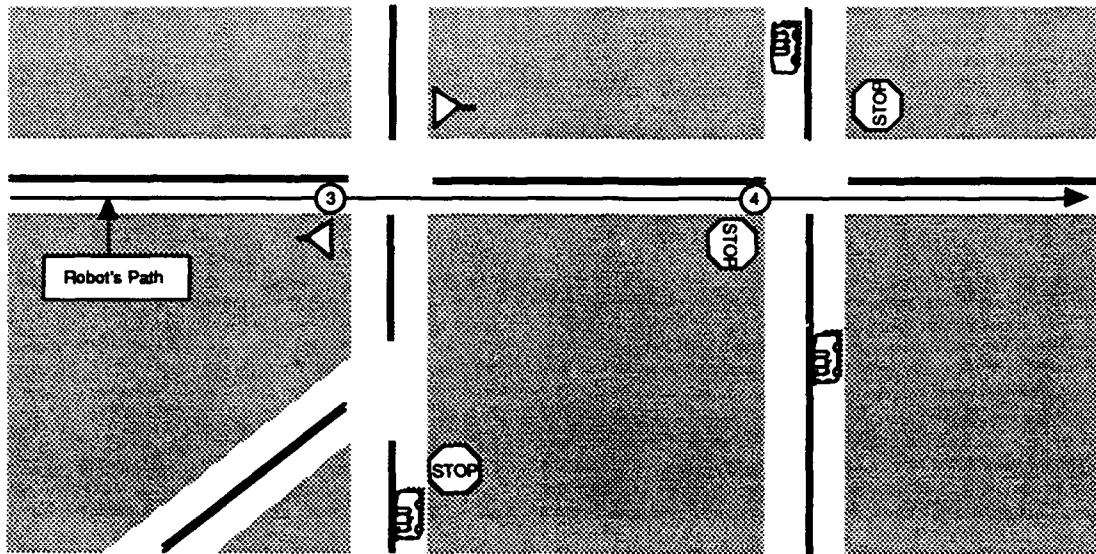


Figure 5-17: Multiple intersection scenario (not to scale).

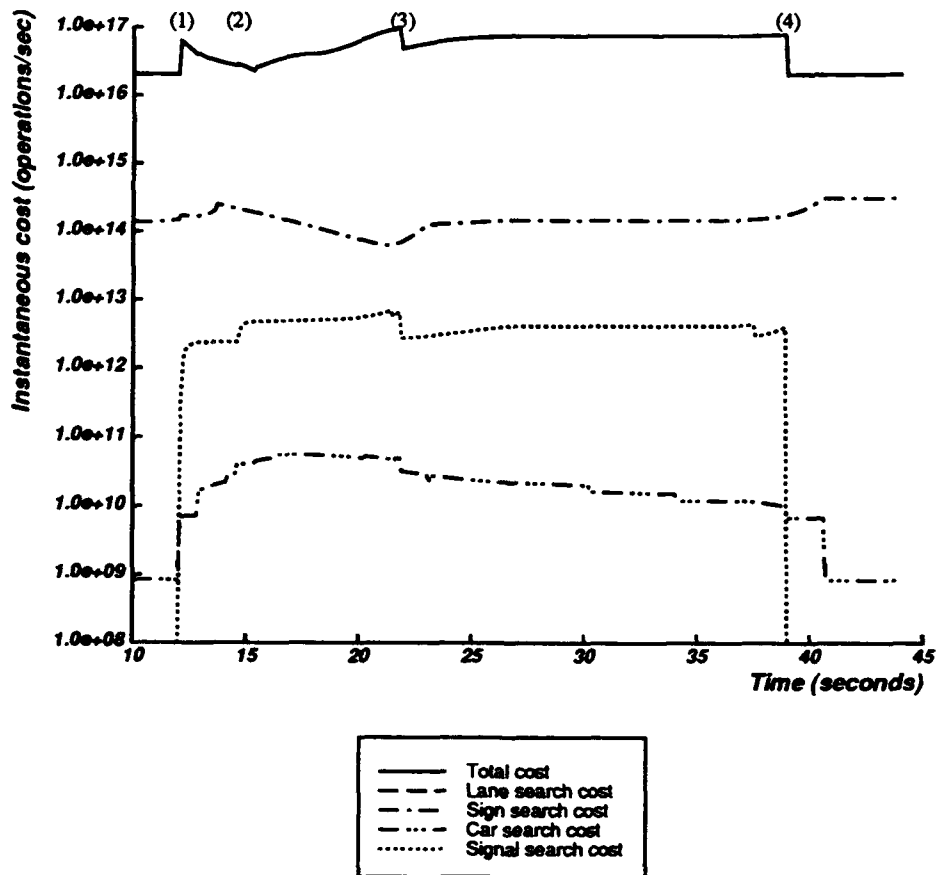


Figure 5-18: Perceptual costs during scenario. Notes correspond to figure above, except: (1) sensors reach first intersection; (2) sensors reach second intersection. Total cost is nearly the same as lane search cost.

Chapter 6

Ulysses 2: Ignoring Redundant Constraints

The previous chapter described how perceptual routines can sharply limit where the perception system has to look for objects in the scene. However, while the routines determine where to look for specific objects, they do not by themselves determine *which* objects must be found. In the Ulysses-1 system, the reasoning component still asks the perception component to look for everything that could possibly generate a constraint or preference. In the second implementation of the driving system, Ulysses-2, the reasoning component requests only enough objects to determine a unique action for the robot. Ulysses-2 takes advantage of the specificity of perceptual routines to find one object at a time. Perceptual routines are requested sequentially in an efficient order.

Figure 6-1 illustrates the ideas behind Ulysses-2. The robot in the figure has not examined

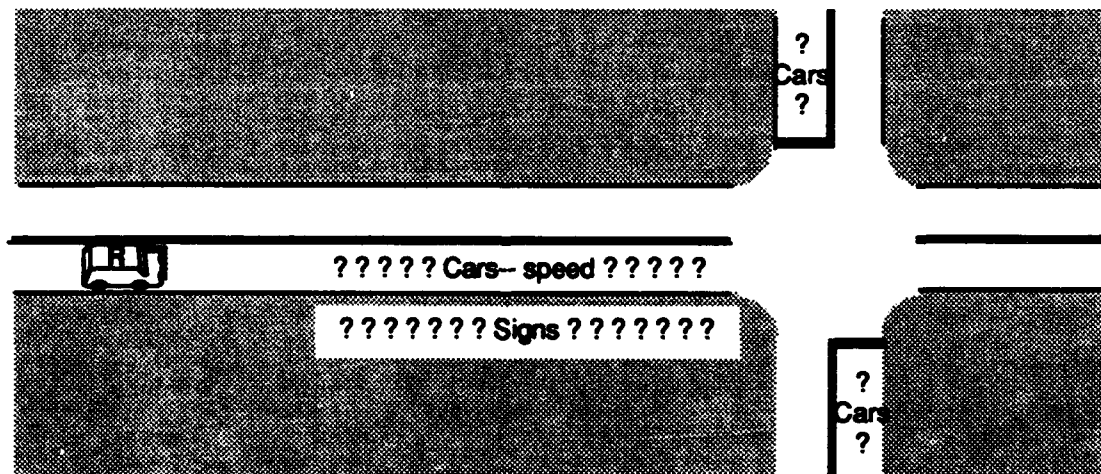


Figure 6-1: Robot that has not yet sensed lead car speed, signs, or cross traffic.

cars in front of it, nor searched for signs along the side of the road, nor looked for traffic approaching the intersection. Each of these could generate constraints on the robot's speed. If we want a conservative, safe robot, then if it is uncertain about the world it should assume very pessimistic conditions. In this example the robot could assume that a car in front of it

in the lane was stopped, that there was a Speed Limit 15 sign just ahead, and that cars were about to cross the intersection from both directions. Since a stopped car in front of the robot causes the worst constraint (lowest allowed acceleration), the robot should look for cars first. Figure 6-2 shows how the criticality of constraints can be determined by plotting deceleration curves on a distance-speed graph. If the robot actually did find a stopped car just ahead, there would be no point in looking at speed limit signs because even a 15mph speed limit could not cause harder deceleration. If there were no slow cars ahead, then the robot should next look for signs, since a nearby 15mph speed limit would require higher deceleration than would a stop at the distant intersection. If there were a slow car ahead, the constraint it generated would depend on its range and speed, and the robot might or might not need to examine the intersection.

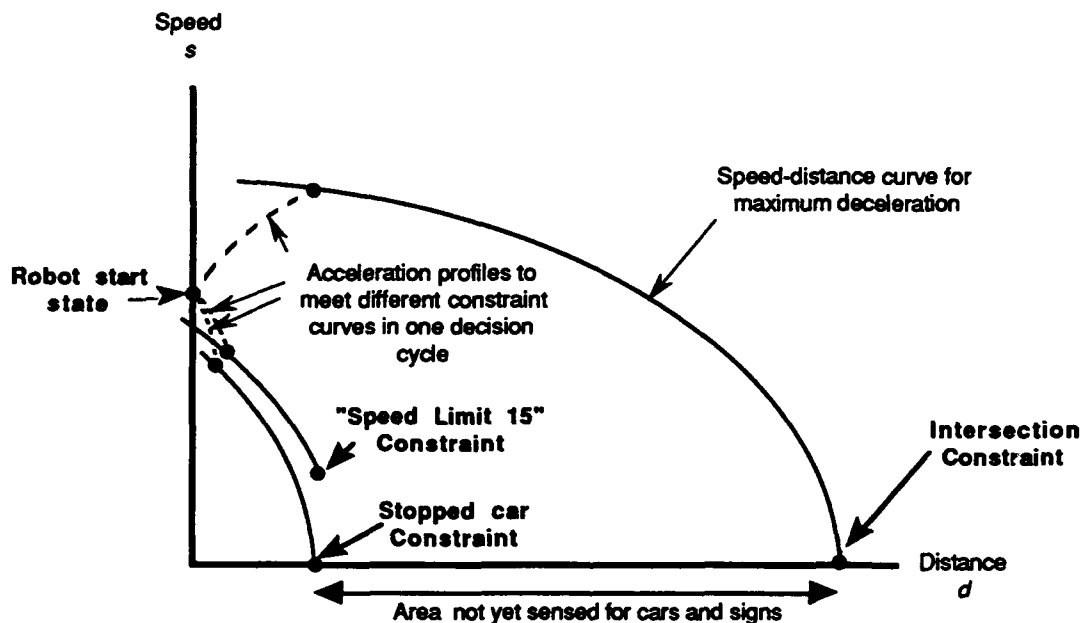


Figure 6-2: Worst-case constraints plotted on a distance-speed graph.

This chapter explains how Ulysses-2 automatically orders the search for traffic objects based on the criticality of constraints. Ulysses-2 uses frame-like data structures to organize objects in the current corridor and specify how to sense them. Driving logic is encoded explicitly in an inference tree structure which captures the uncertainty about yet-undetected objects. Finally, Ulysses-2 uses a branch-and-bound style algorithm to evaluate the tree with as little perceptual cost as possible.

6.1. Ulysses-2 Data Structures

6.1.1. The Corridor

The corridor model in Ulysses-1 is simply a topological description of the lanes and intersection paths and a set of image "markers" to name them. The markers are used as reference points to find markings, signs, signals, etc., which are used to generate constraints and preferences. In Ulysses-2 the corridor is expanded to become a structure with slots for all of the possible objects related to the road. The corridor structure thus explicitly represents all of the objects that are part of the driving model. Figure 6-3 shows parts of this structure.

The figure shows that the corridor can have several major parts, each of which represents a lane in the road or an intersection. The lane parts have slots for cars ahead in the lane, signs along the roadside, over-lane signs, markings, left and right lane lines, and the lane width. Since there can be a variable number of some objects, some slots actually point to lists of objects. Signs, markings, and cars are structures themselves, with slots for type (signs and markings), distance, speed (for cars), etc. There are also pointers to adjacent lanes and to the corridor parts ahead and behind. Intersection parts have slots for traffic control signs, signals, cars in the intersection, and approach roads. Approach roads are also structures, and contain slots for lane markers, backward-facing traffic signs, and approaching cars.

As Figure 6-3 and the above description indicate, the corridor is a hierarchy of structured objects. At the bottom of the hierarchy are individual facts, encoded in structures called *data items*. The data items have three attributes: a value, a status, and an update program. The value is the sensed value of that item. The status is either "new" or "current;" "new" indicates that the value *has not yet been sensed* and is therefore *unknown*. The update program specifies how perceptual routines are to be used to sense a value. (In this way, the corridor data structure is like a frame [Minsky 75]. At the beginning of each decision cycle, a corridor data structure is created with all of the slots for objects already in place, but with the data items marked "new." Thus the driving logic starts with no information about what objects actually exist and must run the update programs for individual data items to sense the actual situation.

The corridor structure may change and grow during the course of a decision cycle. In fact, at the start of the decision cycle, the only part that exists is a special start marker obtained by looking directly in front of the robot. Thus the known world ends only a short distance away. However, a special status slot indicates that the world ends because it has not yet been sensed, not because the end was sensed. This status slot has an update program that

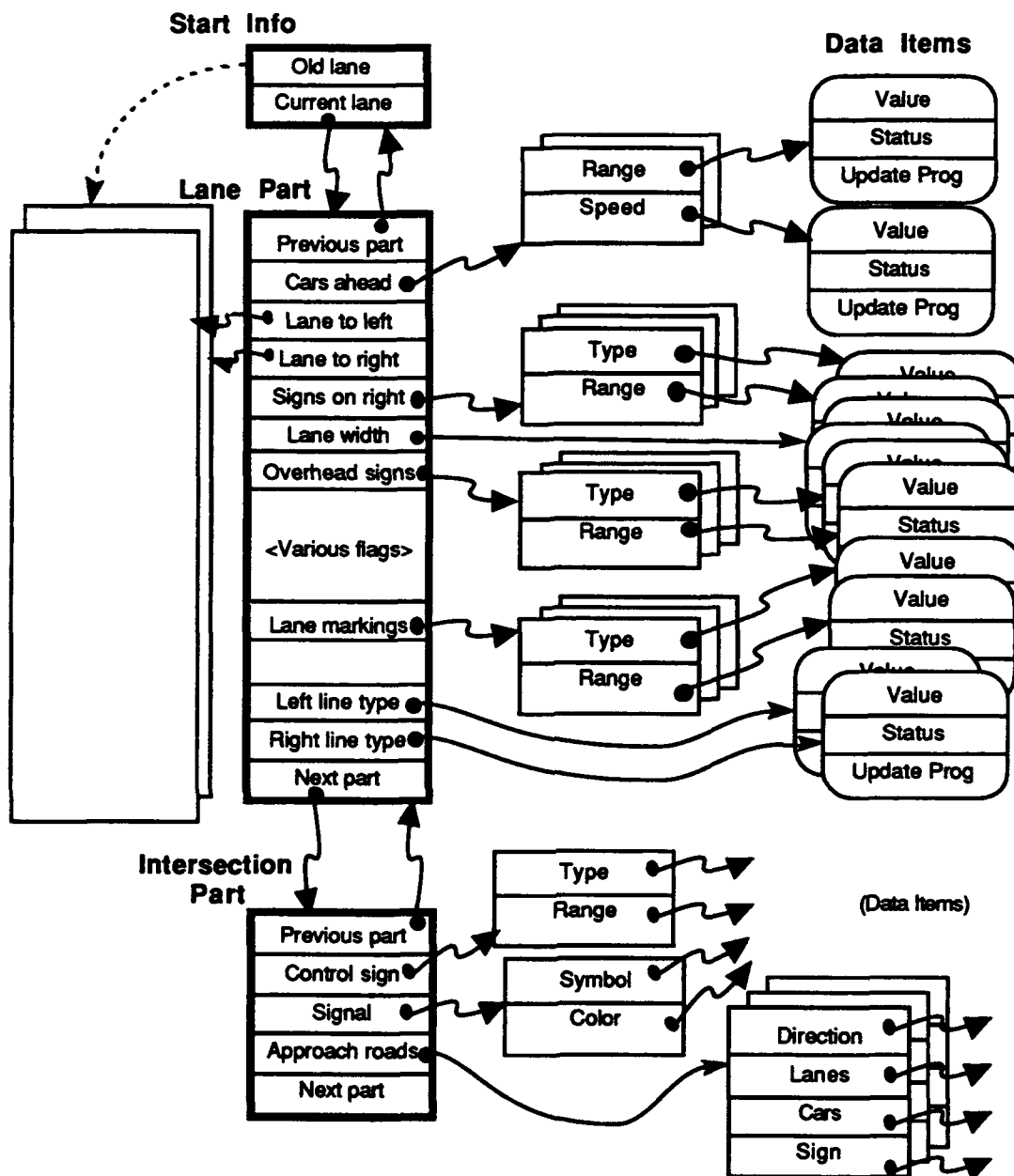


Figure 6-3: The Ulysses-2 corridor model.

requests perception to look ahead for the next part of the corridor. Similar mechanisms are used to grow the next segment of the corridor, and lists of signs, markings, approach roads, and cars.

6.1.2. The Inference Tree

Ulysses-2 uses an inference tree to explicitly represent the Ulysses driving knowledge. With this explicit representation, Ulysses-2 is able to reason about how sensory data affects robot actions and select the best fact to sense. This section describes the inference tree data structure.

The constraint and preference values used to determine the robot's actions are computed by applying various numerical and symbolic functions to sensor data. Ulysses-1 performs this processing with embedded programs. In Ulysses-2, each of the functions is represented explicitly as a *node*. Figure 6-4 shows the schematic of a node. A node has a list of inputs, an output value, and a function definition. The nodes are linked together into an inference tree that computes constraints and preferences and then combines them to determine an action. This tree is similar to the inference tree that is described by if-then rules in an expert system.

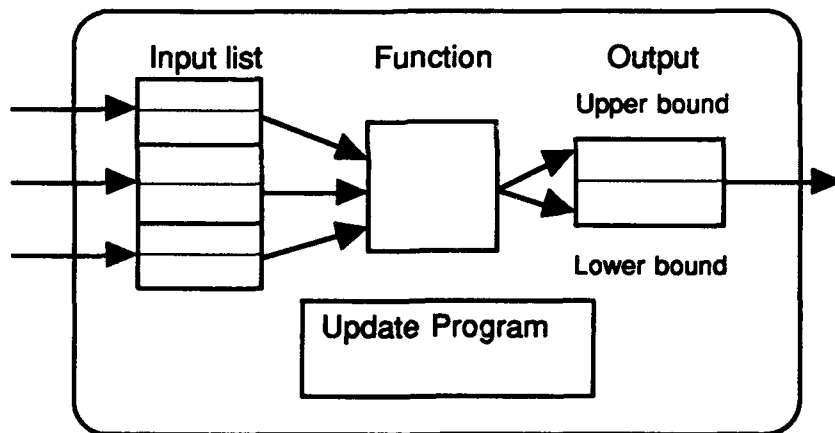


Figure 6-4: A Ulysses-2 node.

The figure indicates that the output of a node is allowed to have a range of values, specified by an upper and lower bound. We can thus also think of the inference tree as a one-way constraint propagation network, with bounds on the inputs to the leaf nodes being propagated to the root node through the node functions. The inputs to the leaf nodes are sensory data, so the inference tree in effect translates a range of possible world states into a range of robot actions. The goal of Ulysses-2 is to find the minimum set of objects to sense that will reduce the uncertainty in the root node to one unique action.

Symbolic values. Ulysses-2 uses boolean and symbolic values as well as numbers in the inference tree. Boolean values fit well into the scheme of bounded intervals because they can

take only two values. When a boolean value is uncertain, its bounds are True and False. Other symbolic values must be used in special, restricted ways for intervals to be meaningful. For example, the symbolic values can be *sets*, with the restrictions that *Lower bound* \subseteq *Upper bound* and that "increasing" the lower bound can only be done by adding elements to the set. Ulysses-2 uses symbol values as inputs to boolean functions (for example, set membership and equivalence) and for the *ad-hoc* generic functions described in section 6.2.1. Appendix D describes the use of symbolic values in more detail.

Sense nodes. The leaves of the tree are special *sense* nodes. These nodes are each assigned to a data item in corridor structure, which defines the object or value they sense. Several different sense nodes may be assigned to the same data item. An initialization program in each sense node defines the initial range of values the node output should have when the data item is "new," i.e. has not yet been sensed. This is where knowledge about characteristics of the environment is explicitly encoded. The initial node bounds must be worst-case, so that when the data is sensed the value will fall between the bounds. It is possible to initialize these bounds to a small range if the right task knowledge is available. In general though, there are few limits Ulysses-2 can use on completely unknown objects. For example, the maximum speed of a car might be limited to 100mph, and the minimum set to 0. Figure 6-5 illustrates how the corridor structure and the inference tree are connected through sense nodes and data items.

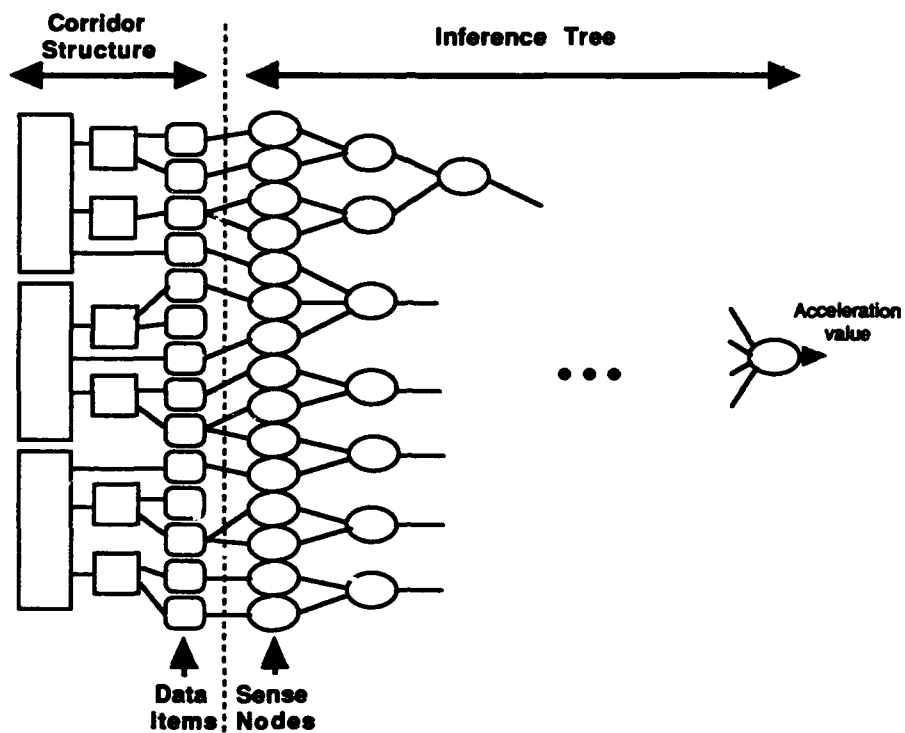


Figure 6-5: The corridor is connected to the inference tree via sense nodes

Tree building. The inference tree is dynamic, much as the corridor structure is. As new objects are found by perceptual routines, new subtrees are opened to add constraints and preferences. The tree modification is performed by **update programs** attached to nodes (see Figure 6-4). The update programs are run whenever new values are propagated up through a node. The corridor and tree growing effects will be described in more detail in the next section.

6.2. The Search Algorithm

In each decision cycle, Ulysses-2 finds the corridor and invokes constraints and preferences corresponding to each corridor part (lane or intersection). The computations for the constraints and preferences are described by subtrees of nodes. Constraints and preferences are "invoked" by generating these subtrees and combining them together. The result is an inference tree that describes how objects around the corridor could determine the robot's action. After building a new tree, and before requesting any sensing actions, Ulysses-2 initializes the sense nodes to worst-case bounds and determines what robot actions are allowed as a result. This process of tree creation and propagation of bounded values is described in section 6.2.1 below.

The next step in the decision cycle is to determine a unique robot action. This step requires that Ulysses-2 determine the actual value of some sense nodes by sensing. Ulysses-2 selects the best sensing action by searching the inference tree. The search procedure starts at the root and descends towards the leaves. At each node, Ulysses-2 must choose one input (i.e., tree branch) to explore. Ulysses-2 chooses the input that currently determines the bounds of the node's value. For example, for a node that computes the Maximum of its inputs, the input with the highest upper bound is chosen. The procedure repeats until a sense (leaf) node is reached; Ulysses-2 then requests a sensing operation to determine a value for this node. After the sensing operation, the new value is stored in the corridor and also propagated back up the tree. The search is repeated until there is a unique action at the root. This search algorithm is described in section 6.2.2.

6.2.1. Tree Creation and Bounds Propagation

At the beginning of every decision cycle, Ulysses-2 grows new inference trees and initializes the bounds of each node by propagating values from the leaves to the root. Ulysses-2 makes two trees—one for acceleration decisions and one for lane decisions. The node descriptions and mechanisms for growing the trees are contained in a Node Description List and an Inference Engine Module (IEM). The Node Description List defines all node types by listing their names, functions, inputs, and update programs, if any. The IEM defines an Open procedure and the bounds propagation mechanism for each type of node function.

Creating nodes. The Open procedure is used to create constraints and preferences when Ulysses-2 needs them for a new corridor part. Each constraint and preference has a corresponding node in the Node Description List. Ulysses-2 calls the Open procedure on this node (and recursively on its inputs) to build the tree that computes the constraint or preference. The Open routine comprises the following steps:

```

procedure Open (Node-name)
  create new node structure;
  fill in slots from Node Description List;
  for each input, Open(input);
  initialize bounds of node value;
  link node to parent

```

For sense nodes, the Open process is slightly different. The inputs to sense nodes are not other nodes, but data items from the corridor structure. The sense nodes take their values directly from the data items without Opening any other nodes. Thus the recursive node Opening process stops at the sense nodes. The data items have not yet been sensed when the tree is created, so the sense nodes take their *initial* values from their own initialization programs (described in section 6.1.2). Figure 6-6 illustrates the entire Open process on a simple subtree.

Propagating bounds. The node bounds (except sense nodes) are initialized using propagation mechanisms defined in the IEM. The IEM includes formulas to propagate bounded values through every node function in the inference trees. All of these formulas are described in detail in Appendix D. The IEM currently handles about two dozen *key* functions, plus routines for *generic* and *special* functions. The *key* functions include Min and Max, predicates of numbers ($>$, $=$), arithmetic functions ($+$, $-$, \times), boolean functions (and, or, not), set functions (member-p, intersection), and conditionals (if-then-else for numbers and symbols). For example, Figure 6-7 shows how some bounded input values would propagate through three *key* function nodes.

Generic functions are known to the IEM only as, for example "fn-2" or "fn-1-sym" (for a function of two arguments returning a number, and a function of one argument returning a symbol, respectively). The actual function is defined in the Node Description List. Figure 6-8 shows a generic node description and the propagation of intervals through this node. Although these function definitions *could* be almost anything, in Ulysses-2 they are restricted. In particular, functions returning numbers must be monotonic in all of their inputs. Monotonicity is important because the IEM computes the bounds of a node only from the bounds of the inputs. If a minimum or maximum of a function were to occur in the middle of the input range, then the node's bounds would be incorrect. Figure 6-9 illustrates this effect.

Finally, *special functions* are simply *ad hoc* functions that were easier for me to define monolithically than to break down into components. An example is "Accel-eqn," which

Node Description List

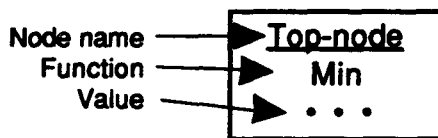
Name: Top-node
Function: Min
Inputs: Eyeball, Ear

Name: Eyeball
Function: Sense
Input: <data item>
Init program: Set bounds to [3, 6]

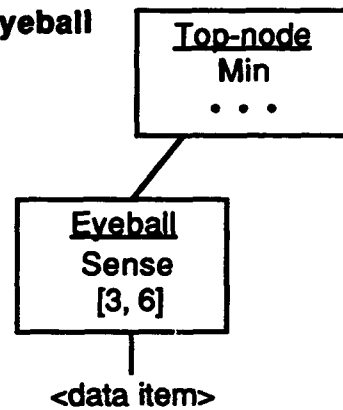
Name: Ear
Function: Sense
Input: <data item>
Init program: Set bounds to [1, 7]

Open (Top-node)

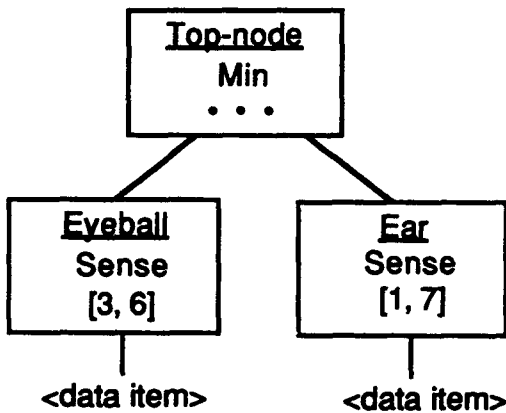
1. Create and Fill Top-node



2. Open Eyeball



3. Open Ear



4. Initialize bounds

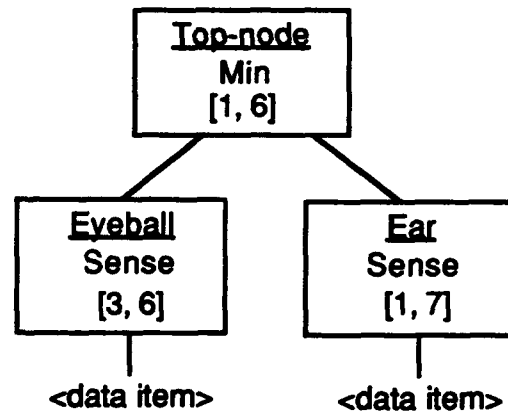


Figure 6-6: Example of recursive Open process, which creates and initializes node trees from information in the Node Description List.

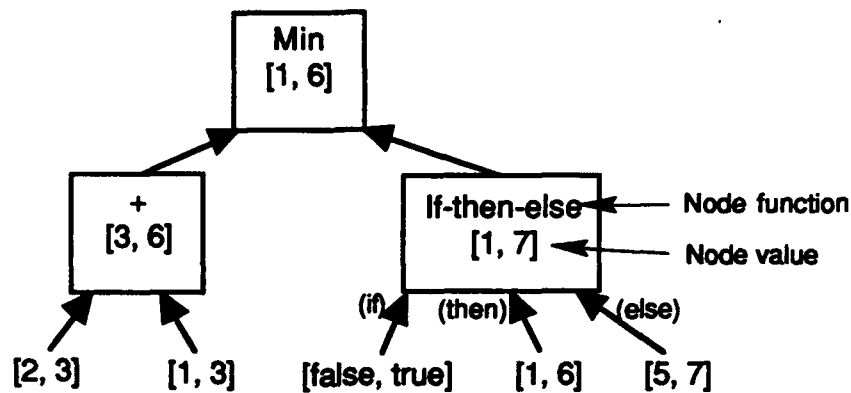


Figure 6-7: Example of interval value propagation through *key* functions. The intervals at the bottom would come from lower nodes in the tree.

Node Description List

Name: My-node
Function: Fn-1
Function df: $f(x) = |\sqrt{x}|$
Inputs: ...

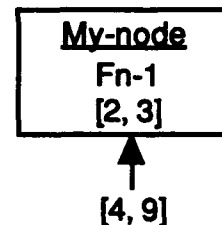


Figure 6-8: Example of interval value propagation through a *generic* function.

computes the acceleration required to meet a constraint (see section 3.3.1), given the robot's speed, the constraint speed, and the constraint range. The IEM knows how to propagate bounds on the input arguments to bounds on the output for this particular function. Special functions must also be monotonic.

Combining constraints and preferences. Chapter 3 explained how acceleration and lane selection decisions are made by constraining possible actions, and then choosing among available actions with prioritized preference rules. These constraints and preferences can be combined with Min and Max functions. For example, acceleration constraints can be combined easily with a Min node. As explained in Chapter 3, acceleration constraints generate values that are upper limits of allowed acceleration; an acceleration is chosen by taking the minimum of all of these upper limits. Figure 6-10 shows the top of an acceleration

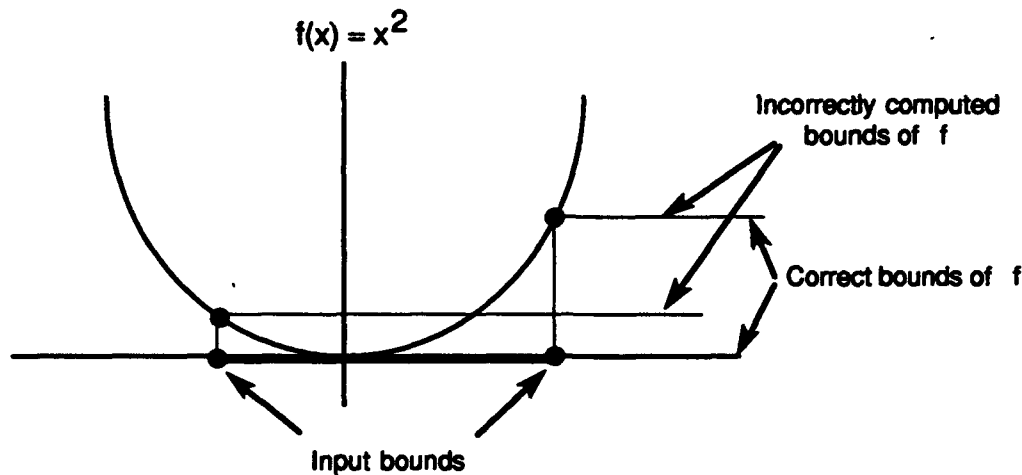


Figure 6-9: Example of errors possible in computing bounds of a non-monotonic function just from the bounds on its input.

inference tree, taken from the example of Figures 6-1 and 6-2. Whenever a new acceleration constraint is generated during a decision cycle, the corresponding subtree is added to the input list of the top level node described in the figure. Note that when the root node of the acceleration tree is uncertain, its bounds reflect the bounds of this upper acceleration limit, not the lower and upper acceleration limits.

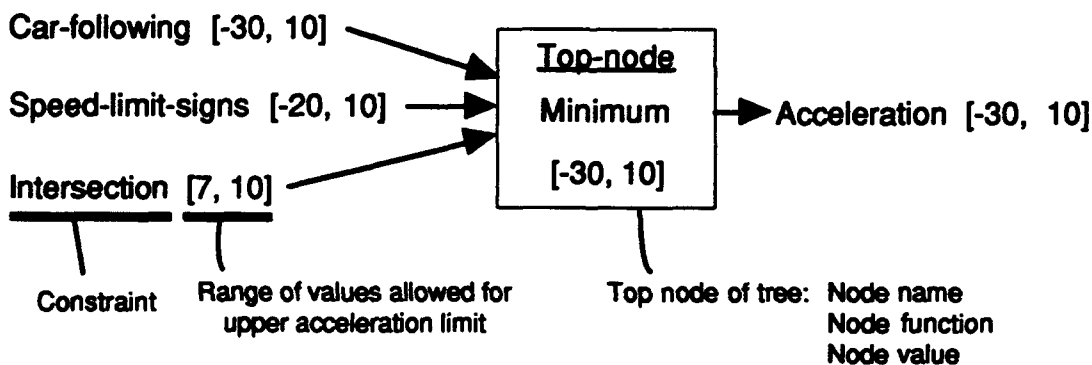


Figure 6-10: The top of an acceleration inference tree. Interval values are indicated by brackets, i.e. [

Lane selection is more complex, and more general, because the actions are symbolic and

there are explicit preference rules. Since the preference rules have a priority ordering, they are assigned number which reflects this ordering (higher numbers indicate higher priority). The top node in the lane selection inference tree first takes the maximum of the priority values, and then looks up the action corresponding to the rule with that priority. This Lane-selection function is described in more detail in Appendix D.

6.2.2. Tree Evaluation

After Ulysses-2 creates an inference tree, it must perform sensing actions to determine the actual value of the top node. When the inference tree is first created, all node values are intervals. After some of these intervals have been reduced to single values by sensing, the root node's interval will also collapse to one value. The goal is for Ulysses-2 to do as little sensing as possible while evaluating the root node.

The basic evaluation process can be described as follows:

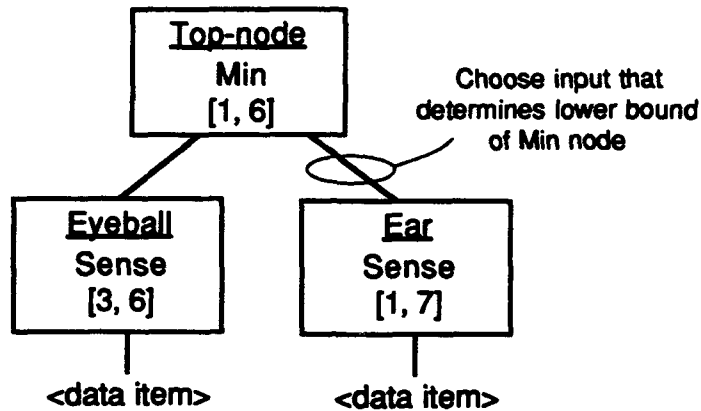
```

procedure Evaluate (node)
  while (node upper bound  $\neq$  node lower bound):
    choose the most critical input of the node;
    Evaluate (input);
    update node bounds;
  
```

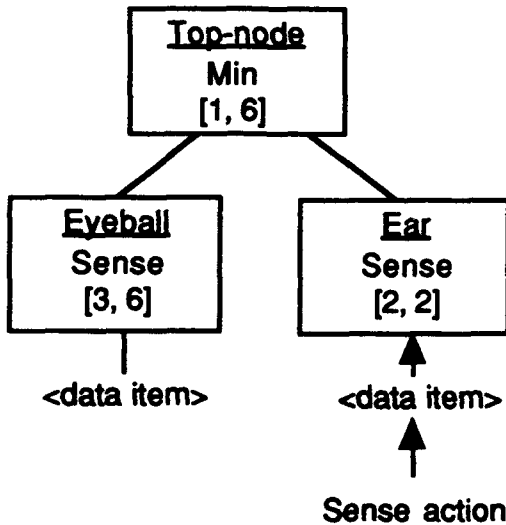
Ulysses-2 calls Evaluate on the root of the inference tree. The descent into the tree ends at the sense nodes, where perceptual actions are performed. Figure 6-11 illustrates how the subtree of Figure 6-6 might be evaluated using this procedure. As explained later in this section, the actual Ulysses-2 evaluation procedure also incorporates cutoffs and iterative search. The final step of Evaluate, "update node bounds", is done with the same bounds propagation mechanism as was used for initialization. "Choosing the most critical input" is the step that determines how much sensing will be required to evaluate the node.

Choosing the right input. In general it is very difficult to choose the right input node for evaluation so as to guarantee minimal sensing cost. Consider the case of a ">" node with a bounded interval for each input. Either input could be bounded more tightly by a sensing action in a data item lower in the tree. When the ranges of the inputs no longer overlap, the node value is uniquely determined. In order to determine which of the N descendant data items should be updated to evaluate the ">" node with minimal cost, Ulysses-2 would have to search through the power set of the data items—i.e., 2^N possibilities. The problem is even more complex because the sensing actions interact with each other, so the *sequence* of sensing actions is also important. Thus for each distinct set of M sense actions, $M < N$, we must also consider the $M!$ possible sequences. Finally, for this to be possible at all, the search algorithm would need an accurate estimate of the cost of each sensing operation before performing the action. Such an estimate is difficult to obtain because the cost of routines depends on the situation in which they are applied—the range, how much visual search is done before an object is found, etc.

1. Choose input of Top-node



2. Evaluate Ear



3. Update bounds of Top-node

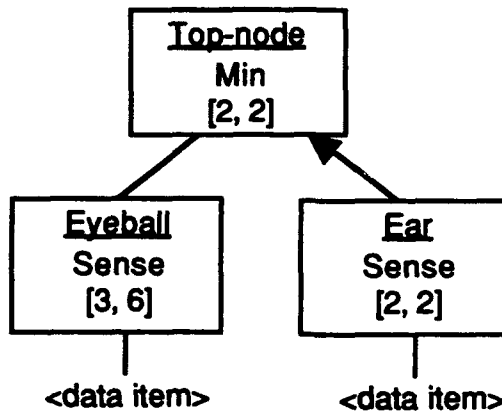


Figure 6-11:

Example of Ulysses-2 tree evaluation. Step 1 requires the selection of the most critical node; node Ear is chosen because the ultimate value of Top-node could be determined by Ear even if the value of Eyeball was known. In step 2 a sense action, specified by the data item, is run to evaluate Ear. In step 3 this node is propagated up to Top-node. The evaluation ends here because the interval in Top-node has collapsed to one value.

Ulysses-2 does not attempt to compute sensing cost estimates and search for a minimal solution in the combinatorially large search space. This is something of a concession to the idea that reasoning could be the limiting factor in system performance. However, in practice it is possible to achieve high efficiency even without finding the optimal sensing strategy. Two important node functions, Min and Max, allow critical inputs to be uniquely determined without any cost computations at all. (The special function "monotonic set intersection" also allows ordering; see Appendix E.) For example, of the inputs to a Max node, the one with the highest upper bound should be evaluated first because it *must* be evaluated before the node's value can be determined. Likewise, the input to a Min node with the lowest lower bound should be evaluated first because the value of the node cannot be determined with certainty before this extreme case is examined. Since the top nodes of both the acceleration and lane choice trees use Min and Max functions (for selecting the lowest acceleration limit and highest priority lane preference, respectively), large portions of the inference tree can be effectively ordered by criticality. For other node functions, Ulysses-2 arbitrarily chooses the first [yet unexplored] input to evaluate next. No additional mechanism is provided to use heuristics to guide in the selection. Appendix E describes the input selection logic for all functions defined in the IEM.

Cutoffs. The example of the ">" node above shows that node inputs do not always need to be determined exactly before the node itself can be evaluated. If the input ranges to ">" do not overlap when the node is created, then the initialization process will collapse the node's bounds to one value. During tree evaluation we would also like to cease the evaluation of the subtree under the node as soon as the input ranges are distinct. This kind of search control is made possible by using *cutoffs*. Cutoffs have been used for some time in branch-and-bound algorithms to search trees composed of Min nodes, Max nodes, or both. In Ulysses-2, the IEM *generates* and passes down cutoff values not only from Min and Max nodes, but also from predicate functions (e.g. "=", "eql", "member-p" and ">"). Cutoff values are *used* not only in Min and Max nodes, but by all numerical functions and the set intersection function. These cutoff values are used in the Evaluate function as follows:

```

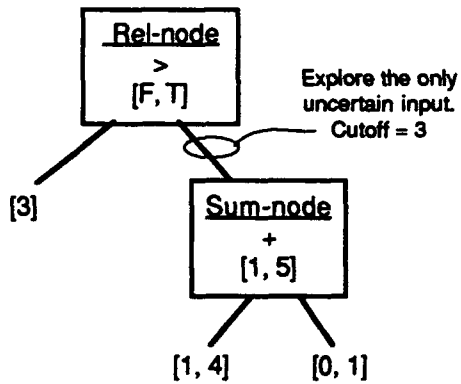
procedure Evaluate(node, parent-cutoff)
  while (node upper bound  $\neq$  node lower bound AND
        node value has not reached parent-cutoff):

    choose the most critical input of the node;
    generate cutoff value;
    Evaluate (input, cutoff);
    update node bounds;

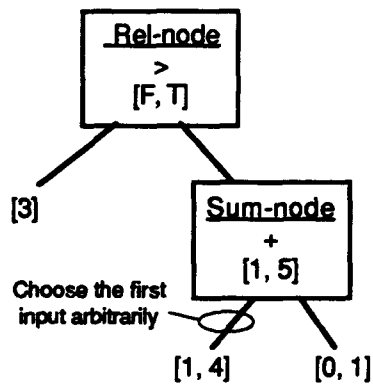
```

There are actually three ways that the cutoff is used: as a maximum value; as a minimum value; and as both. Different node functions generate different types of cutoffs. For example, a Min node generates a maximum value for its children, while an "=" node generates a cutoff that is both. Figure 6-12 shows how a ">" node creates a minimum-value cutoff and how a "+" node uses it. Appendix E describes how cutoffs are generated for all functions.

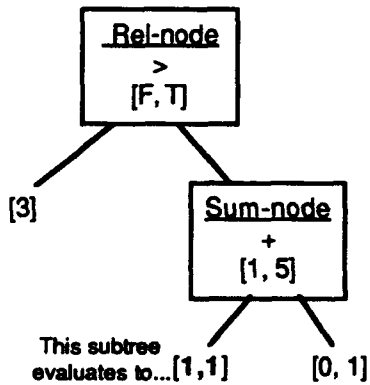
1. Choose input, generate cutoff



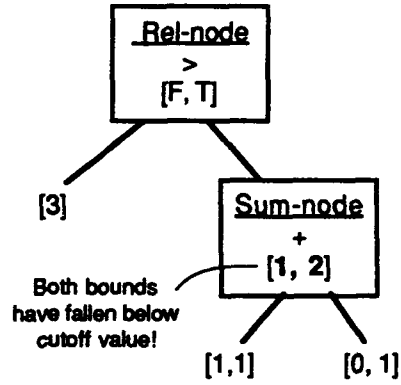
2. Evaluate Sum-node—choose input



3. Evaluate chosen subtree



4. Propagate new value



5. Propagate Sum-node value

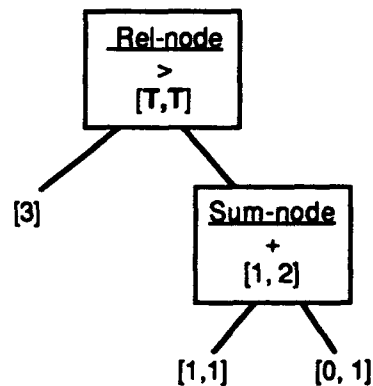


Figure 6-12: Example of cutoff generation and use in Ulysses-2 tree evaluation.

Factors preventing optimal selection. While the recursive Evaluate function described above shows the general process of inference tree evaluation, it is not adequate for Ulysses-2. In the Ulysses-2 inference tree, several facts conspire to prevent the above function from working effectively. First, there are interactions between sensing actions. That is, the sensing program for one data item may actually update several data items. Second, as mentioned earlier, several sense nodes may be tied to the same data item. Third, node update programs may dynamically modify the inference tree node evaluation. Thus after a data item's update program is run it is necessary to update the bounds of all sense nodes that are tied to all data items that are actually updated. When any of these nodes changes, it is necessary to update its parent's bounds, and its parents, etc. until either the root is reached or a node's bounds do not change. At this point, the inference tree may have changed at any level; therefore, rather than recursively popping up a level, the algorithm must start over again from the top. Otherwise, in the worst case, the tree evaluation could be complete and the algorithm would still be considering sense nodes at the bottom! Ulysses-2 meets all of these requirements by using the following *iterative* evaluation function:

```

procedure Evaluate-tree (top-node)
  while (top-node upper bound  $\neq$  top-node lower bound):
    Descend tree from top to a sense node {choosing most
      critical input and generating cutoffs at
      each node};
    Run data item update function;
    For each updated data item:
      Ascend tree until node value doesn't change
      {updating value of each node using a
      cutoff, and running update program,
      if any};

```

This procedure is contained in the IEM.

6.3. Examples

The two examples further explain the operation of Ulysses-2. The first example is a more detailed illustration of the search algorithm and various node functions. The second example shows how trees are created dynamically during the decision cycle. Figure 6-13 shows part of an inference tree for acceleration. Each box is a node, and shows the node name, the node function, and, in brackets, the lower and upper bounds of the node value. In some cases these bounds are symbols, which denote an interval in an ordered set of symbols (see Appendix D for further discussion of symbol intervals). "Accel-eqn" is the function used to compute the acceleration due to a speed constraint at a range (see Section 3.3.1), and "fn-1" is a marker for a generic function that extracts the speed limit from a sign name. This tree might be part of the decision logic for the robot in Figure 6-1. We assume that at this point the robot has already determined that there are no cars in front of it, so it must now consider signs and the intersection. Also, since the intersection has been detected, the robot already knows the range to it.

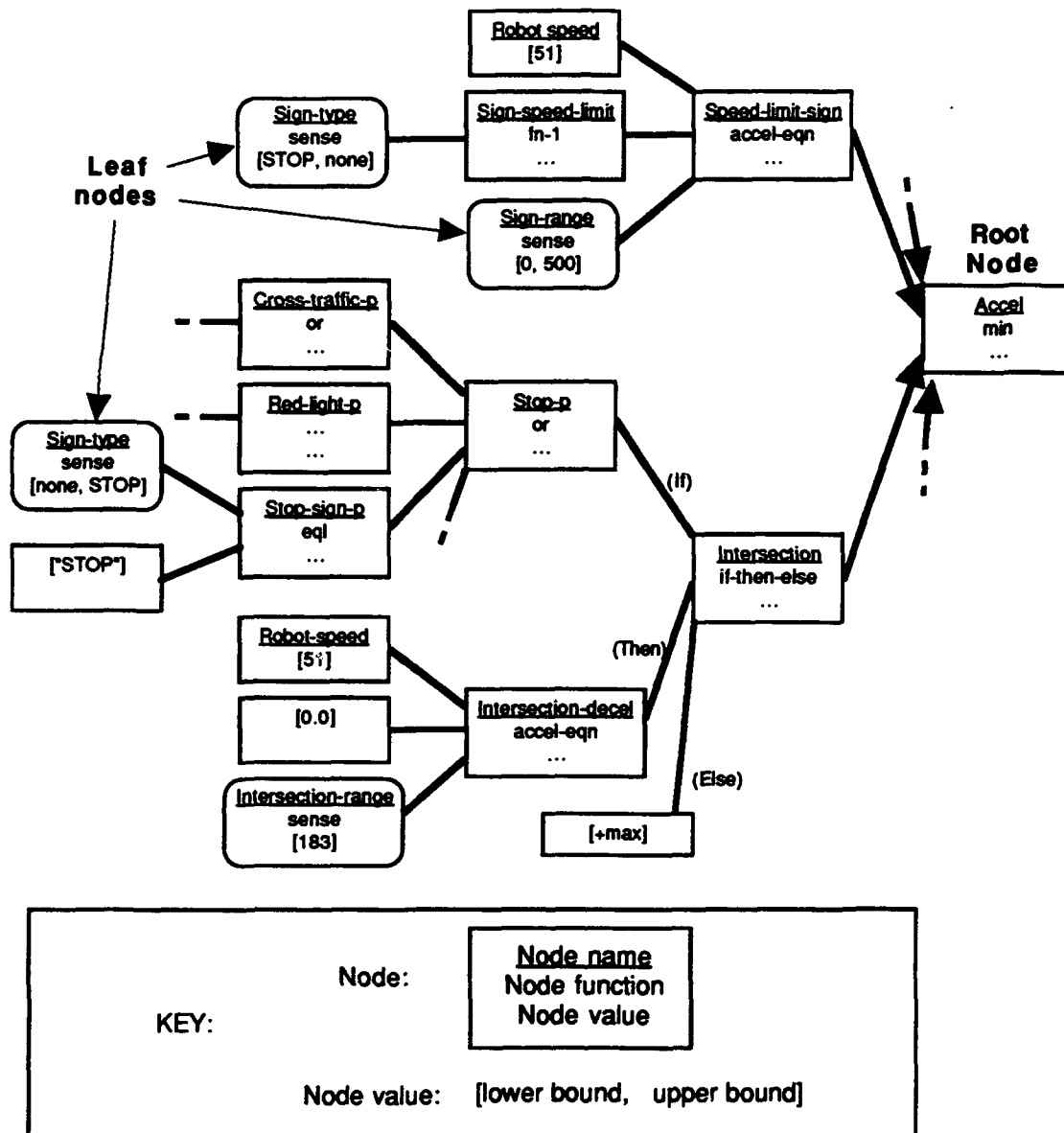


Figure 6-13: Part of the decision tree to determine acceleration for the robot of Figure 6-1.

In Figure 6-13, the tree has been opened and the sense nodes have been initialized with default bounds. The robot speed has already been sensed, so is a single value. Figure 6-14 shows how the default values propagate through the tree to yield a range of allowed acceleration values. "+Max" and "-max" are constants that represent vehicle limits. The upper bound of acceleration is 0 here because the robot is assumed to start at the speed limit. Since the speed-limit-sign constraint is potentially worse than the intersection constraint, Ulysses-2 chooses to look for speed limit signs first. The sense operation for signs scans along the right side of the known road and finds a STOP sign near the intersection. The perceptual routine returns both the sign type and range, so both of the corresponding sense nodes are updated. Figure 6-15 shows the state of the tree after the speed-limit-sign constraint has been updated. The intersection logic uses the same sign, so that sense node is also immediately updated. Figure 6-16 shows the results of backing up the intersection sign node; new values propagate all of the way to the root node, fixing its value and ending the search. Note that if there were another constraint under the top node with a lower bound of less than -4, Ulysses would be forced to explore it too, but could at least set a cutoff value of -4—i.e., if the lower bound of the constraint ever rose above -4, the search would end.

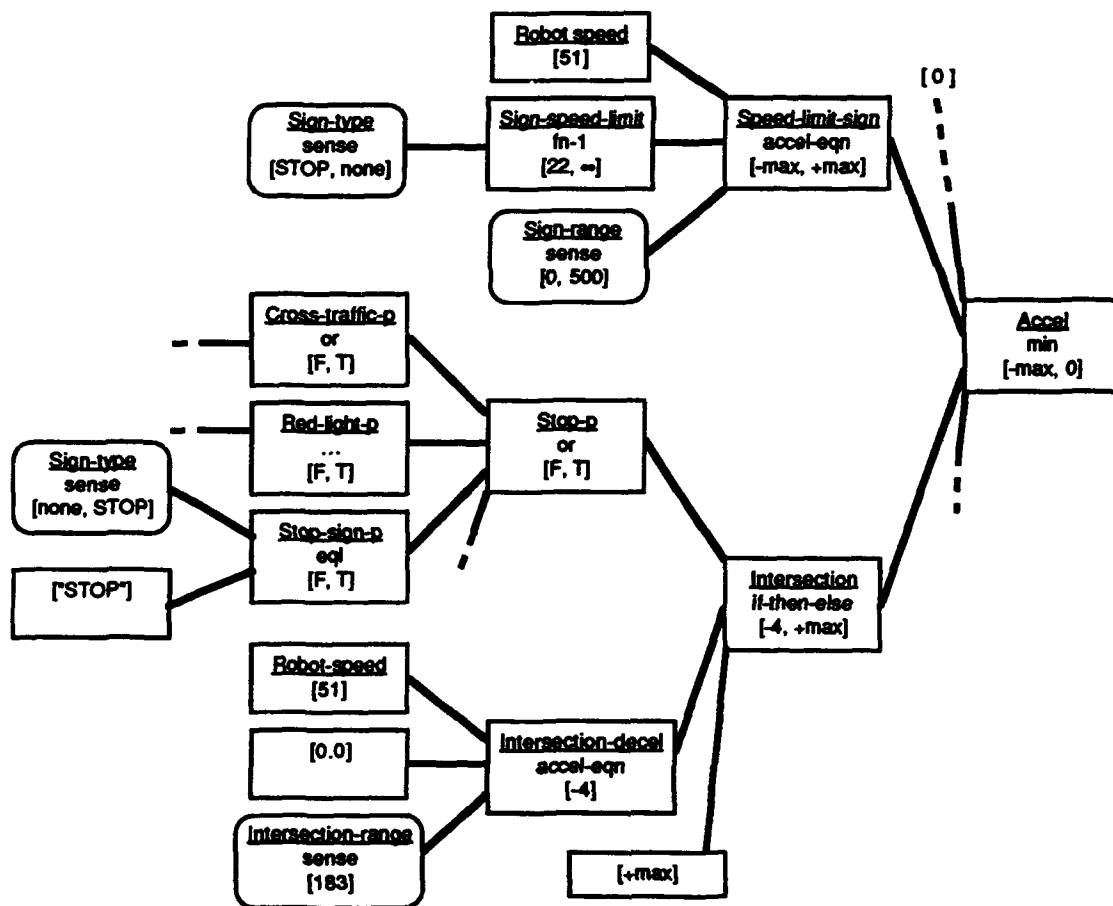


Figure 6-14: The inference tree of Figure 6-13 with initial sense node bounds propagated to the root.

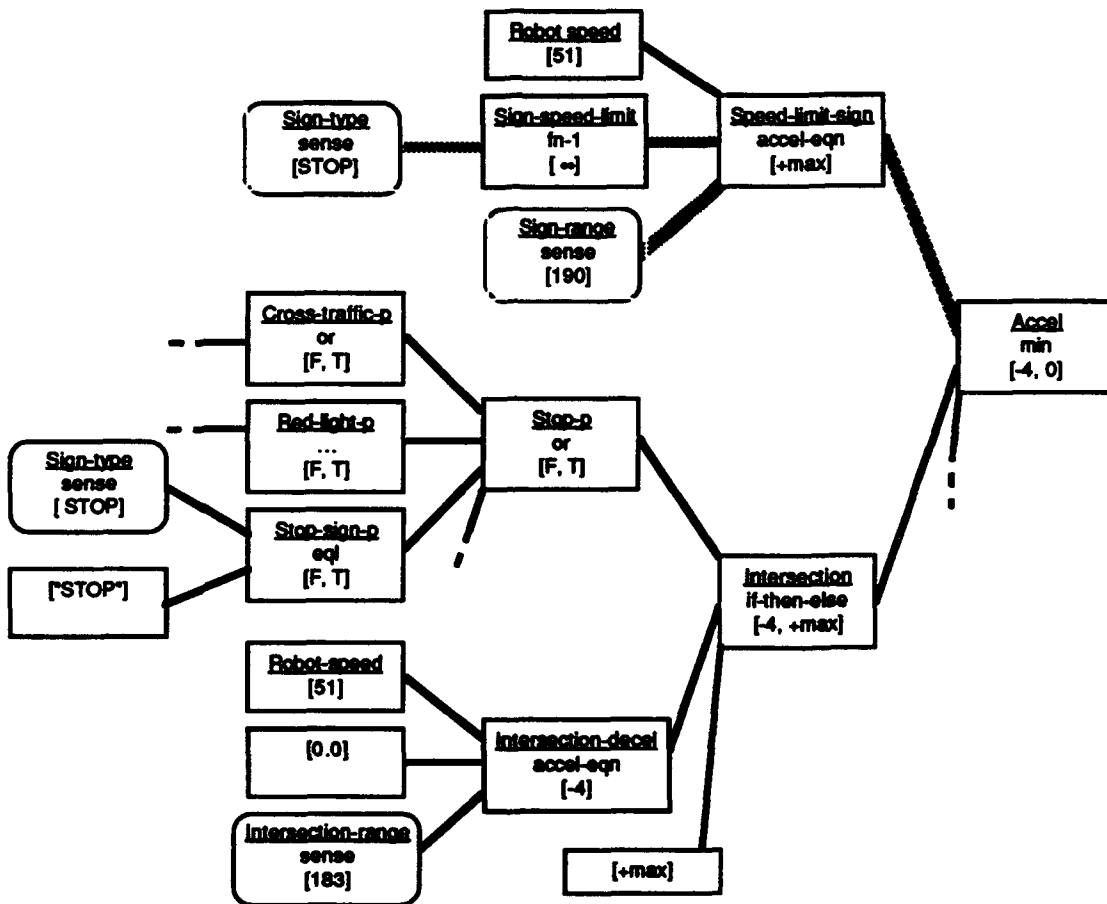


Figure 6-15: The inference tree of Figure 6-14 after signs have been sensed; sensed values have been propagated up the speed-limit-sign subtree.

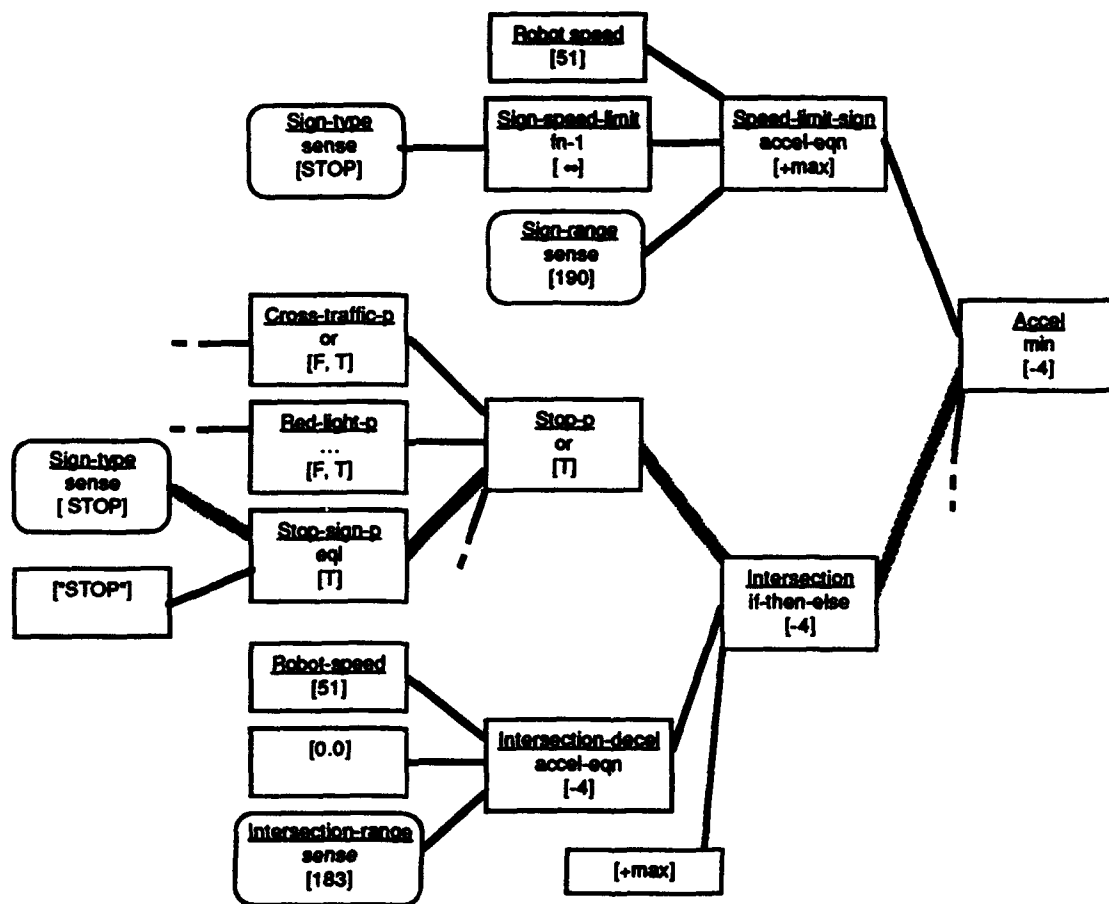


Figure 6-16: Final state of inference tree after sensed sign value has been propagated through the intersection logic.

The second example shows how the corridor and tree are grown dynamically. Figure 6-17 shows one part of the corridor and the corresponding inference tree. In this example the end of the corridor introduces a possible constraint because Ulysses-2 does not know whether the road exists beyond the end. A boolean flag in the corridor structure indicates whether anything else exists, but its status is "new" since the next part has not been sensed. When Ulysses-2 descends the tree to this data item, the item's update program is run and perceptual routines scan the lane beyond the current corridor. When the new corridor part is found, the corridor data structure is augmented as shown in Figure 6-18. The original data item is now "current," and returns the value "True" indicating that the next part does exist. Ulysses-2 backs up this value to the sense node and runs the update program attached to the node. This update program opens a new subtree and attaches it to the existing one, as illustrated in Figure 6-19.

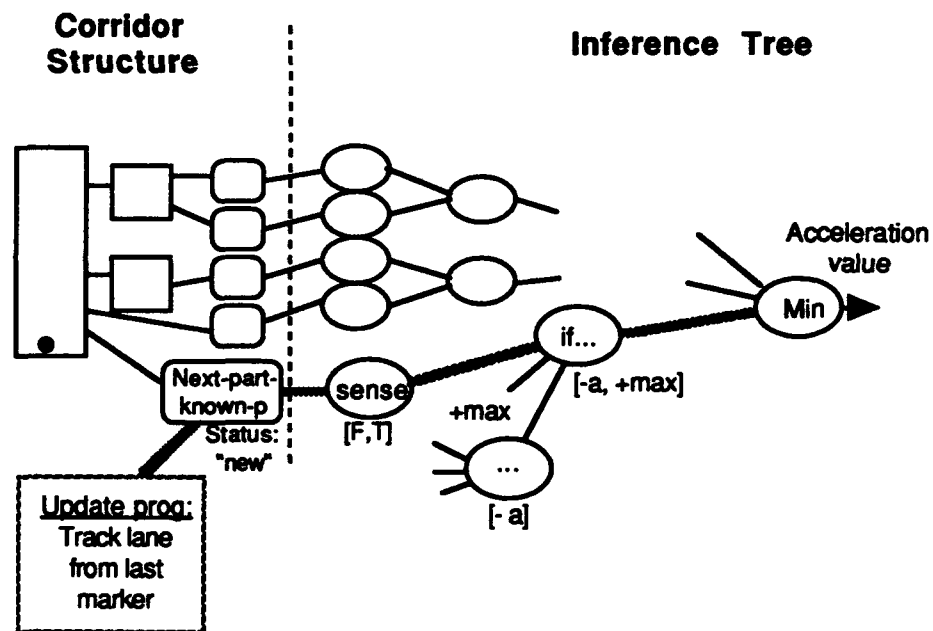


Figure 6-17: The corridor structure with a flag-slot for extending the corridor.

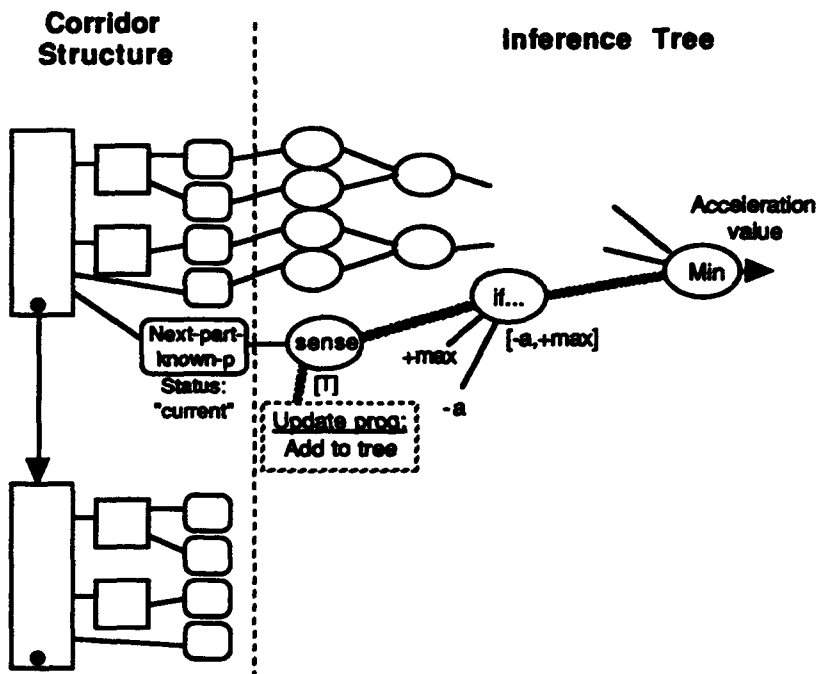


Figure 6-18: New structure added to the corridor as a result of perception.

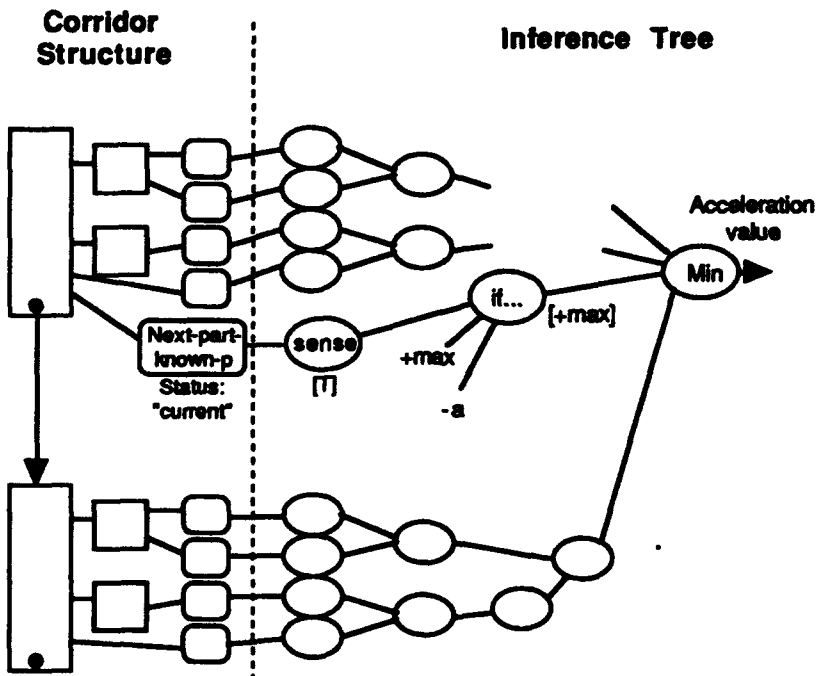


Figure 6-19: Inference tree with new value propagated through, and new subtree added by sense node's update program.

6.4. Experimental Results

The Ulysses-2 system was run on the same set of scenarios used to test Ulysses-1. The intent was for Ulysses-2 to produce the same action decisions as Ulysses-1, but perhaps make fewer perceptual requests in the process. Ulysses-2 is a different implementation of the driving program, so there is no way to guarantee identical behavior to Ulysses-1; however, in all of the scenarios, Ulysses-2 produced the exact same acceleration and lane commands at exactly the same time as Ulysses-1. As the following results show, Ulysses-2 did indeed make fewer perceptual requests.

6.4.1. Left Side Road

Figure 6-20 and Figure 6-21 show the perceptual costs during the left side road scenario. A quick comparison with Figure 5-10 reveals an immediate difference between Ulysses-1 and Ulysses-2: the cost in Ulysses-2 is not always dominated by the cost to search lanes. When the robot is near the intersection at around $47 < t < 49$, the search for signals is most expensive. At this time Ulysses-2 is looking at the other car approaching the intersection, and stops searching for objects when it realizes that this car provides the limiting constraint. A little later, at about $t = 50$, the robot is crossing the intersection and spends most of its effort looking for cars unexpectedly crossing its path. At this point it is going too slowly to bother looking for the lane beyond the intersection.

Figure 6-22 provides a direct comparison with Ulysses-1. Initially Ulysses-2 is cheaper because it does not needlessly examine the adjacent lane for traffic, signs, etc. Unlike Ulysses-1, Ulysses-2 recognizes from the information in its own lane that it will not be necessary to change lanes. However, Ulysses-2 does continually look at the road at the extreme sensor range limit in order to detect the intersection as soon as possible. When the intersection is detected (point 1 in Figure 6-21), there is a spike in the cost as Ulysses-2 considers changing lanes (and decides to do so). The cost then decreases steadily because the robot is getting closer to the intersection and looking at less and less road in front of it. Ulysses-2 does not yet look at the intersection because it would not make any difference in the robot's acceleration decision. The intersection does become critical at around $t = 43$. Figure 6-23 shows this clearly as the lane, car and signal search costs all jump here. At around $t = 47$, the approaching car enters the intersection. Since the intersection is searched before the approach roads, finding a car here allows Ulysses-2 to ignore the approach roads; hence the cost drops dramatically. When the car finally leaves the intersection, Ulysses-2 must search approach roads again until it commits to crossing the intersection (point 5 in Figure 6-22). After crossing the intersection, Ulysses-2 looks for all the same things that Ulysses-1 looks for on the two-lane road.

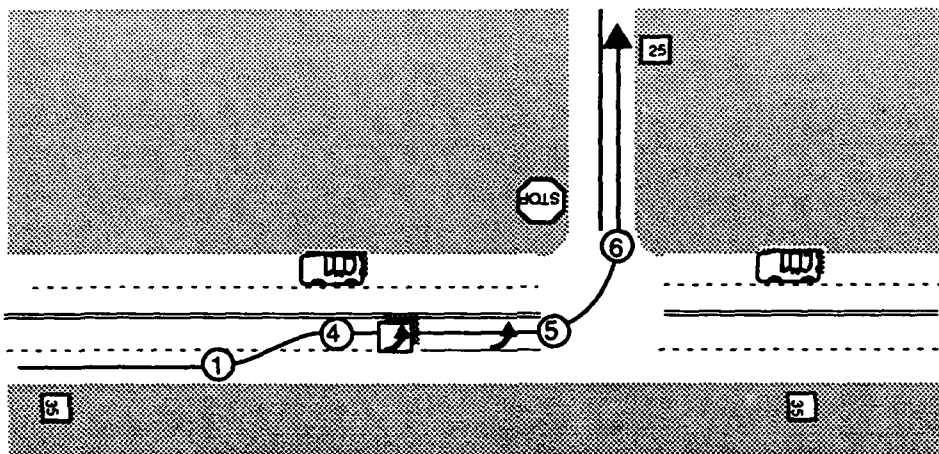


Figure 6-20: Left side road scenario (not to scale).

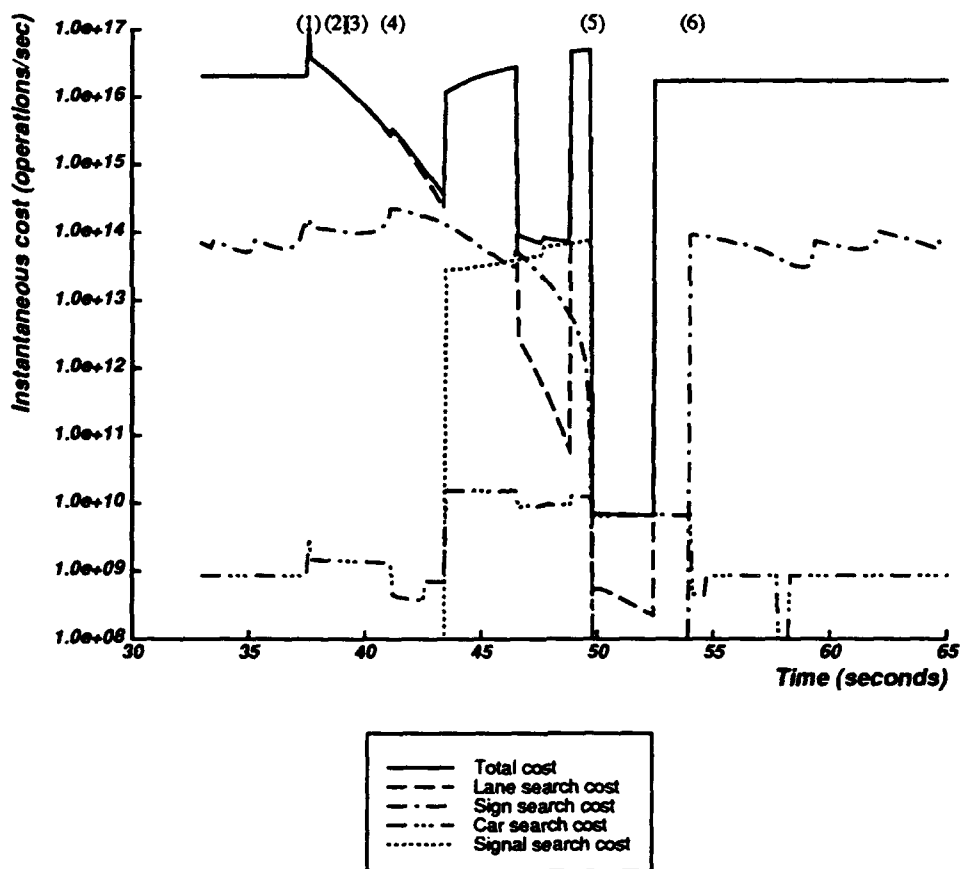


Figure 6-21:

Perceptual costs for Ulysses-2, left side-road scenario. Notes correspond to figure above, except: (2) sensors reach robot's destination street on far side of intersection; (3) sensors reach all intersection approach roads.

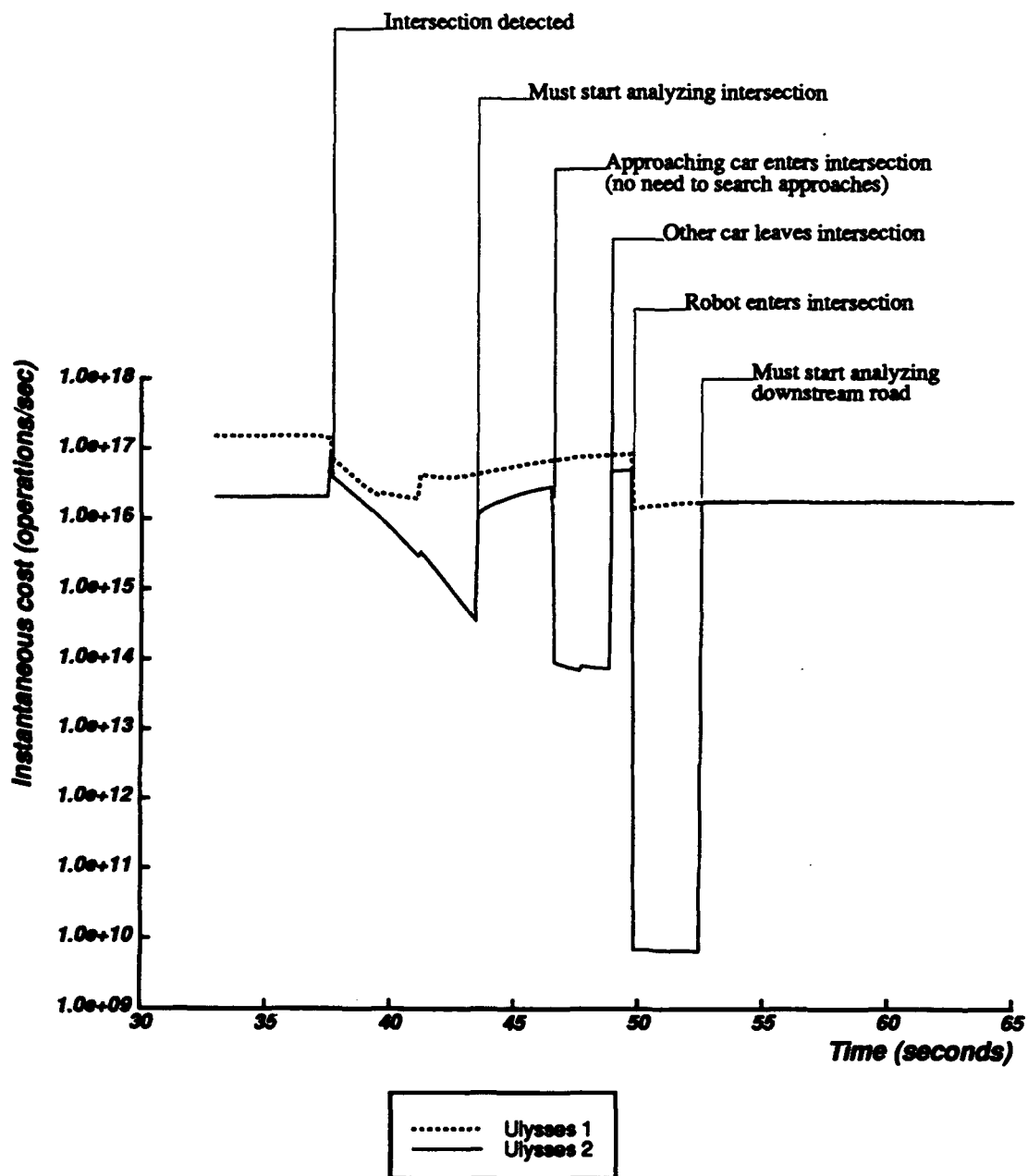


Figure 6-22: Total perceptual cost of Ulysses-2 compared to Ulysses-1 for left side-road scenario.

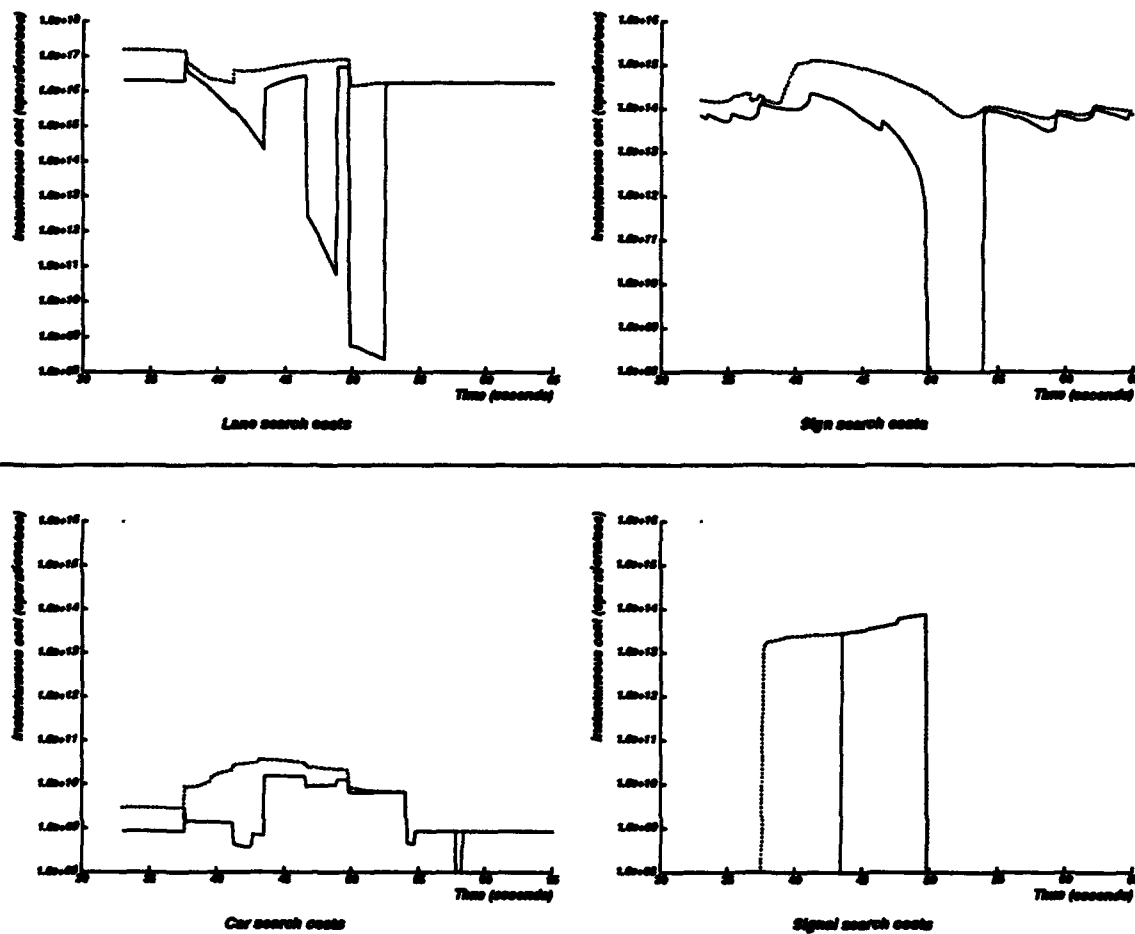


Figure 6-23: Cost of searching for various objects in Ulysses-2 (solid line), compared with Ulysses-1 (dotted line) for left side-road scenario.

6.4.2. Unordered Intersection

Figures 6-24 through 6-27 show the perceptual costs involved in the unordered intersection scenario. The cost plots show many of the same characteristics as the left side road scenario. The robot clearly begins analyzing the intersection at around $t = 16$, as the costs for search lanes, cars, and signals go up. From that point on, Ulysses-2 is looking at both approaching cars. Ulysses-2 looks at *both* because it happens to look left first (but is not constrained by that car). The decision to look left first is arbitrary, because Ulysses-2 contains no knowledge about which approach road might be more likely to generate a constraint first.

At about $t = 23$, the car on the right enters the intersection, and Ulysses-2 no longer needs to search approach roads (as with the previous scenario). This car leaves the intersection after about a second, which forces Ulysses-2 to once again look at all the approaches before committing to the intersection (point 3 in Figure 6-25). Shortly before getting across the intersection, Ulysses-2 begins to scan the next road and the cost goes up again.

6.4.3. Four-lane Highway

Figures 6-28 through 6-31 show the four lane highway scenario in which the robot has to pass a slower car. The graphs show that in Ulysses-2, costs are lower than in Ulysses-1 along a four lane road (as in the left side road scenario above). Ulysses-2 does not examine the adjacent lane until it needs to pass the other car.

6.4.4. Intersection With Traffic Lights

Figures 6-32 through 6-35 show the cost of using Ulysses-2 in the traffic light scenario. Because the light is red as the robot approaches the intersection (points 1 through 6 in Figure 6-33), Ulysses-2 does not bother searching the approach roads at all. Lane searching costs decrease dramatically, and the signal and sign search costs dominate total cost while the robot waits for the light to change. When the light changes at about $t = 92$, lane and car search costs jump up to the Ulysses-1 levels as Ulysses-2 finally takes a look at the other cars. Shortly thereafter the robot enters the intersection and costs drop again as the robot stops looking for anything but unexpected cross traffic. As the robot nears the far side of the intersection, Ulysses-2 again starts to scan the downstream lane.

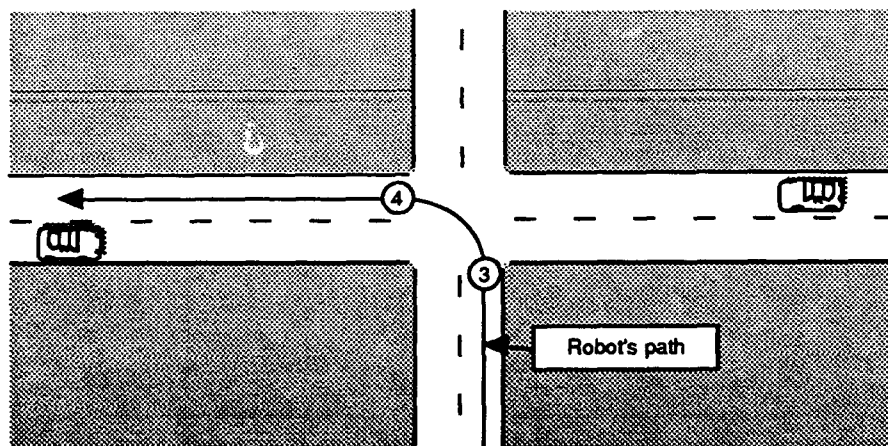


Figure 6-24: Unordered intersection scenario (not to scale).

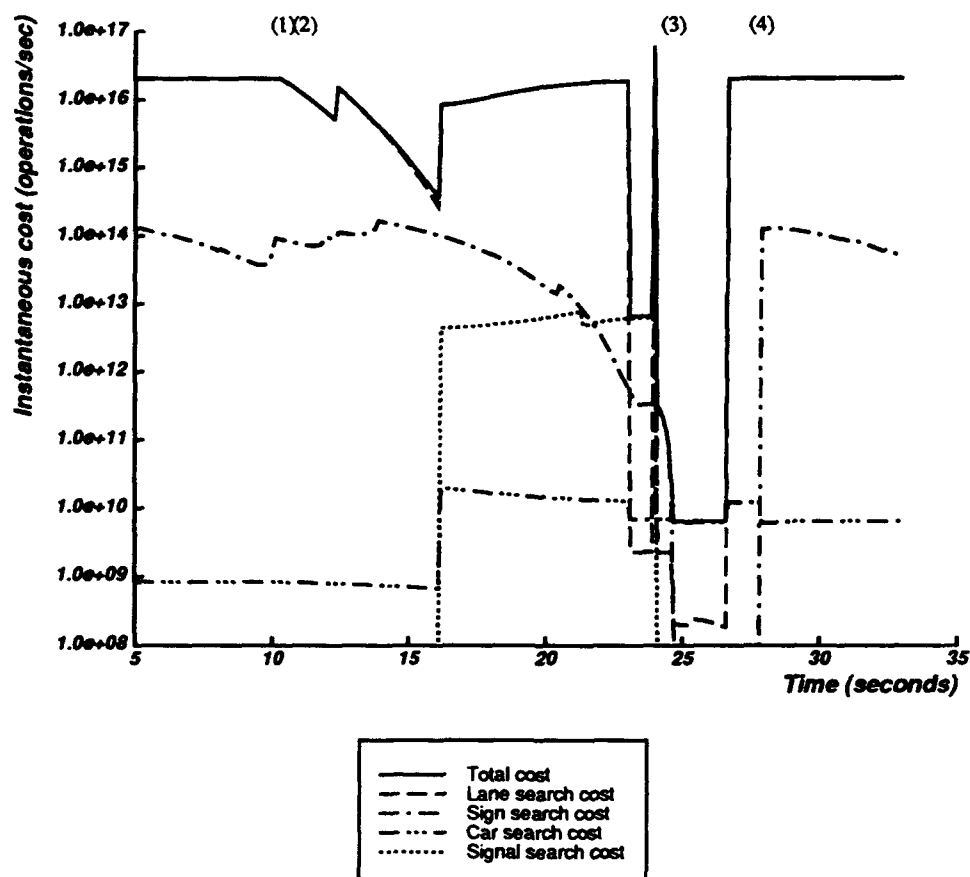


Figure 6-25:

Perceptual costs for Ulysses-2 during unordered intersection scenario. Notes correspond to figure above, except: (1) sensors reach intersection; (2) sensors reach all intersection approach roads.

**THIS
PAGE
IS
MISSING
IN
ORIGINAL
DOCUMENT**

pg 193

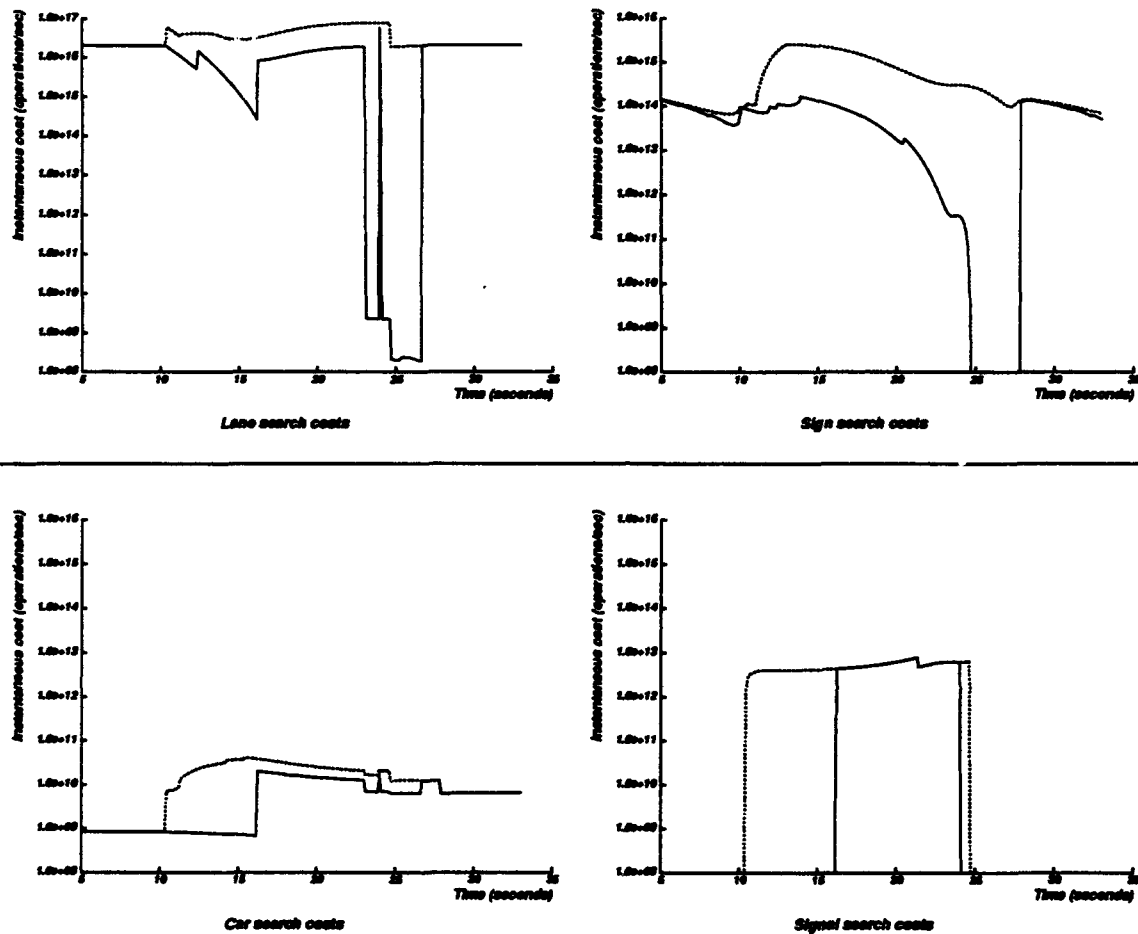


Figure 6-27: Cost of searching for various objects in Ulysses-2 (solid line), compared with Ulysses-1 (dotted line) for unordered intersection scenario.

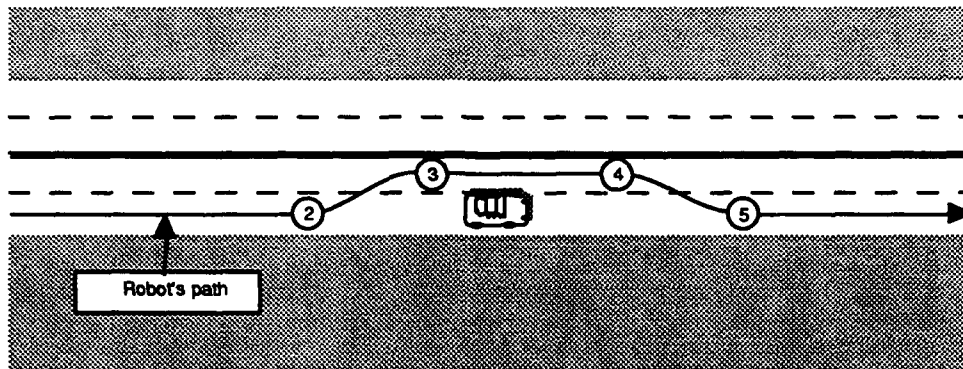


Figure 6-28: 4-lane passing scenario (not to scale).

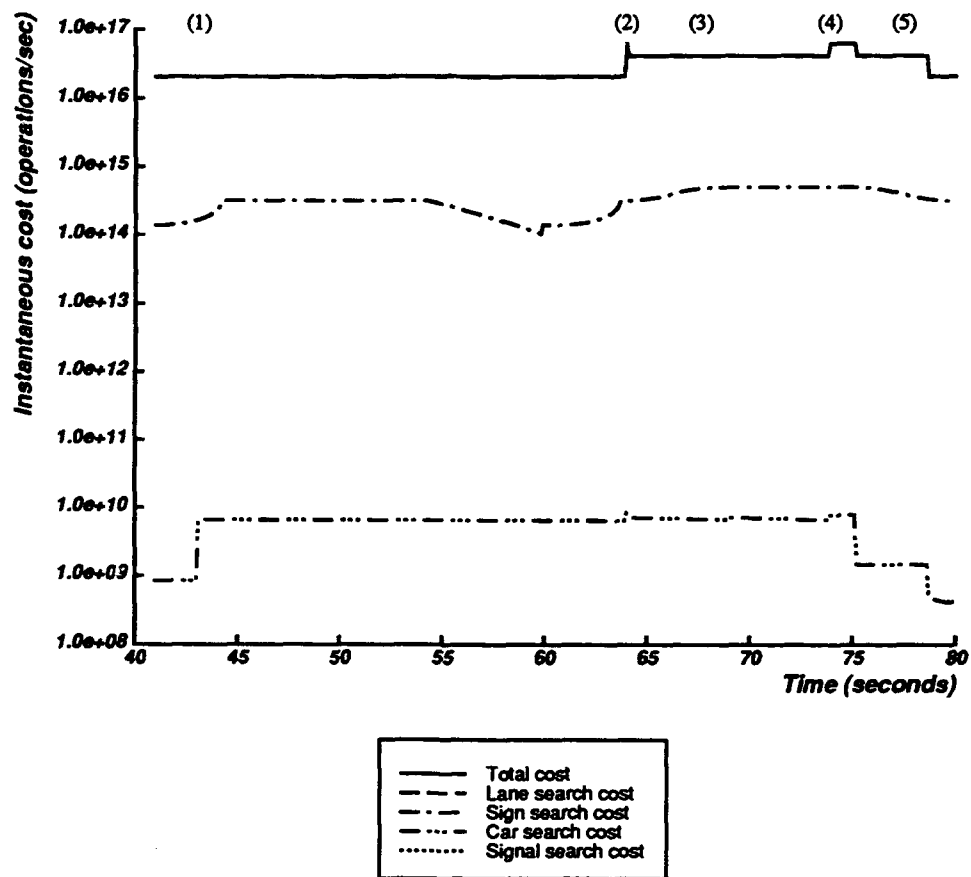


Figure 6-29:

Perceptual costs for Ulysses-2 during passing scenario. Notes correspond to figure above, except: (1) sensors reach lead car.

**THIS
PAGE
IS
MISSING
IN
ORIGINAL
DOCUMENT**

pg 116

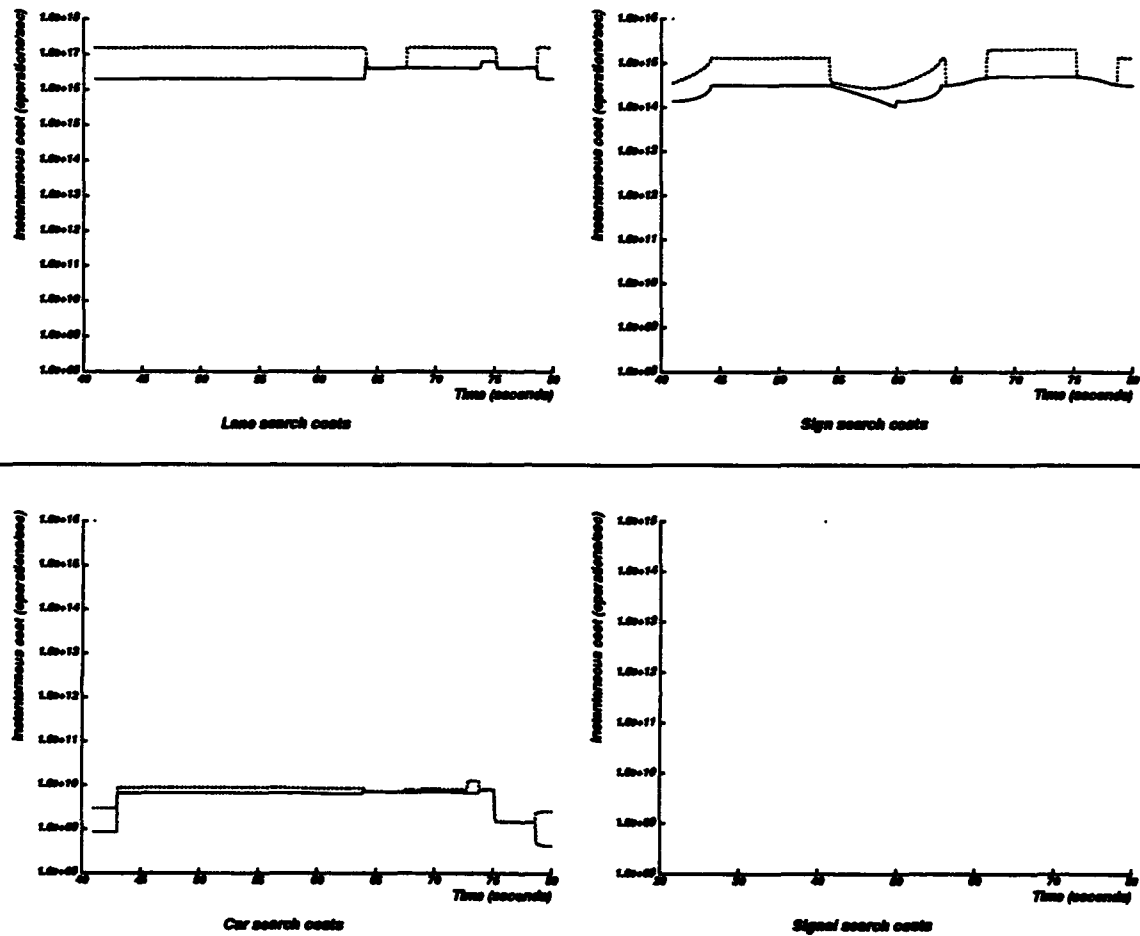


Figure 6-31: Cost of searching for various objects in Ulysses-2 (solid line), compared with Ulysses-1 (dotted line) for passing scenario.

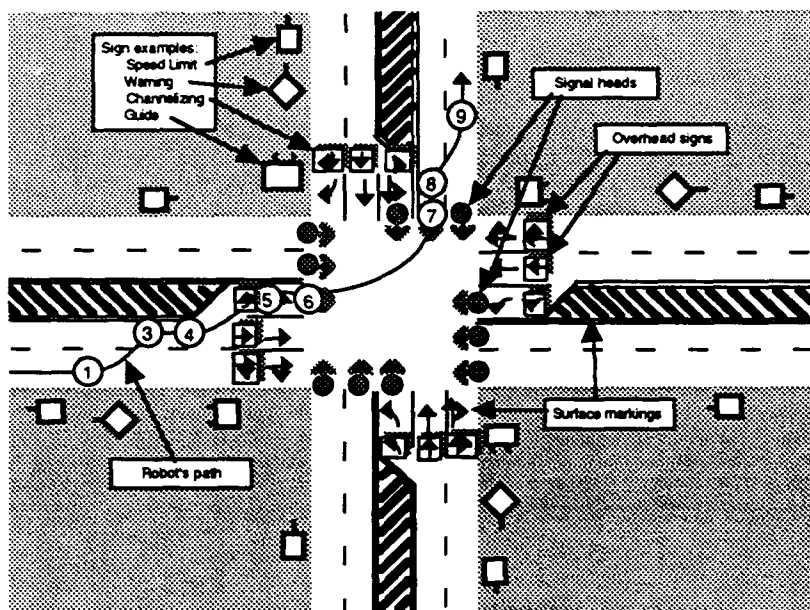


Figure 6-32: Traffic light scenario (not to scale).

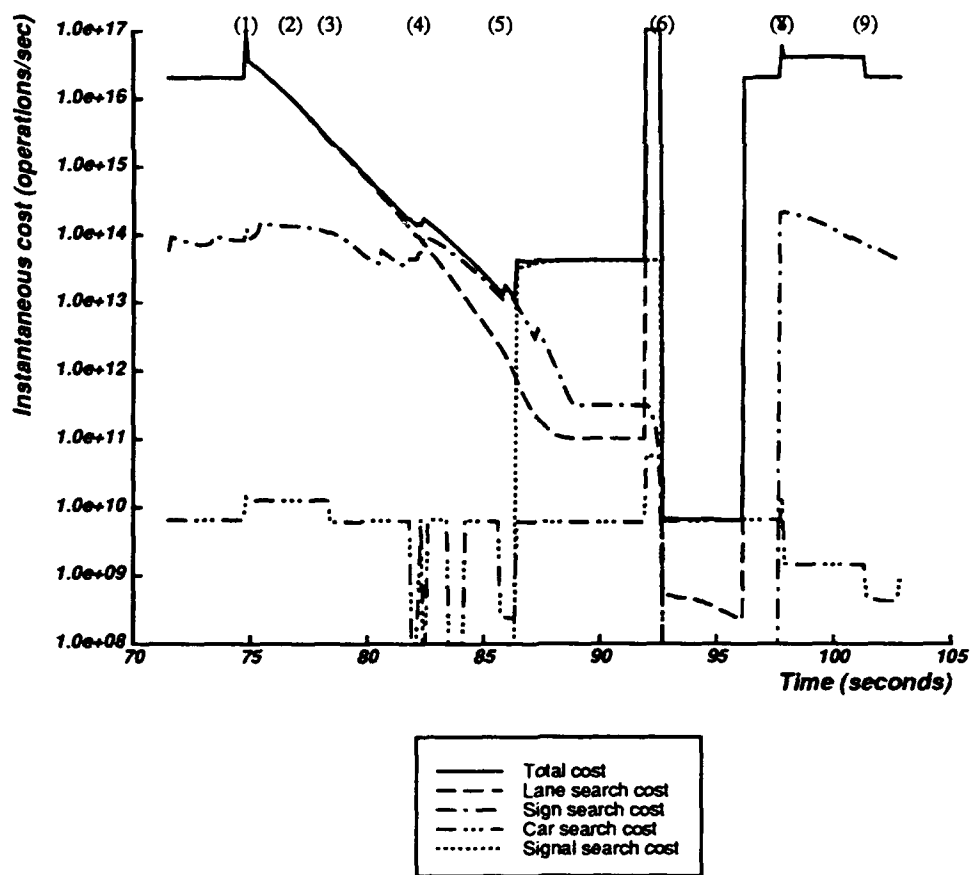


Figure 6-33:

Perceptual costs for Ulysses-2 during traffic light scenario. Notes correspond to figure above, except: (1) sensors reach intersection; (2) sensors reach all intersection approach roads.

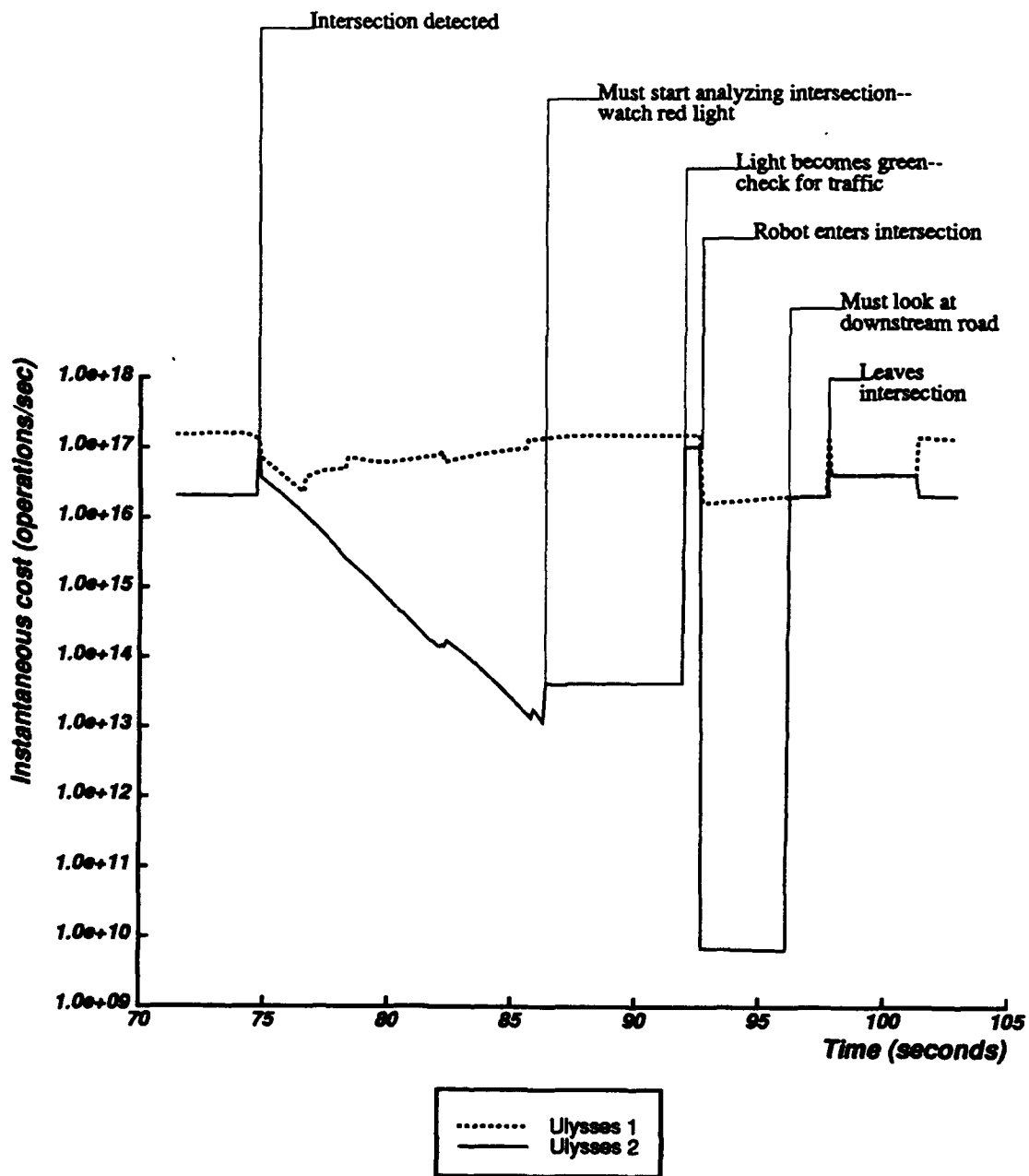


Figure 6-34: Total perceptual cost of Ulysses-2 compared to Ulysses-1 for traffic light scenario.

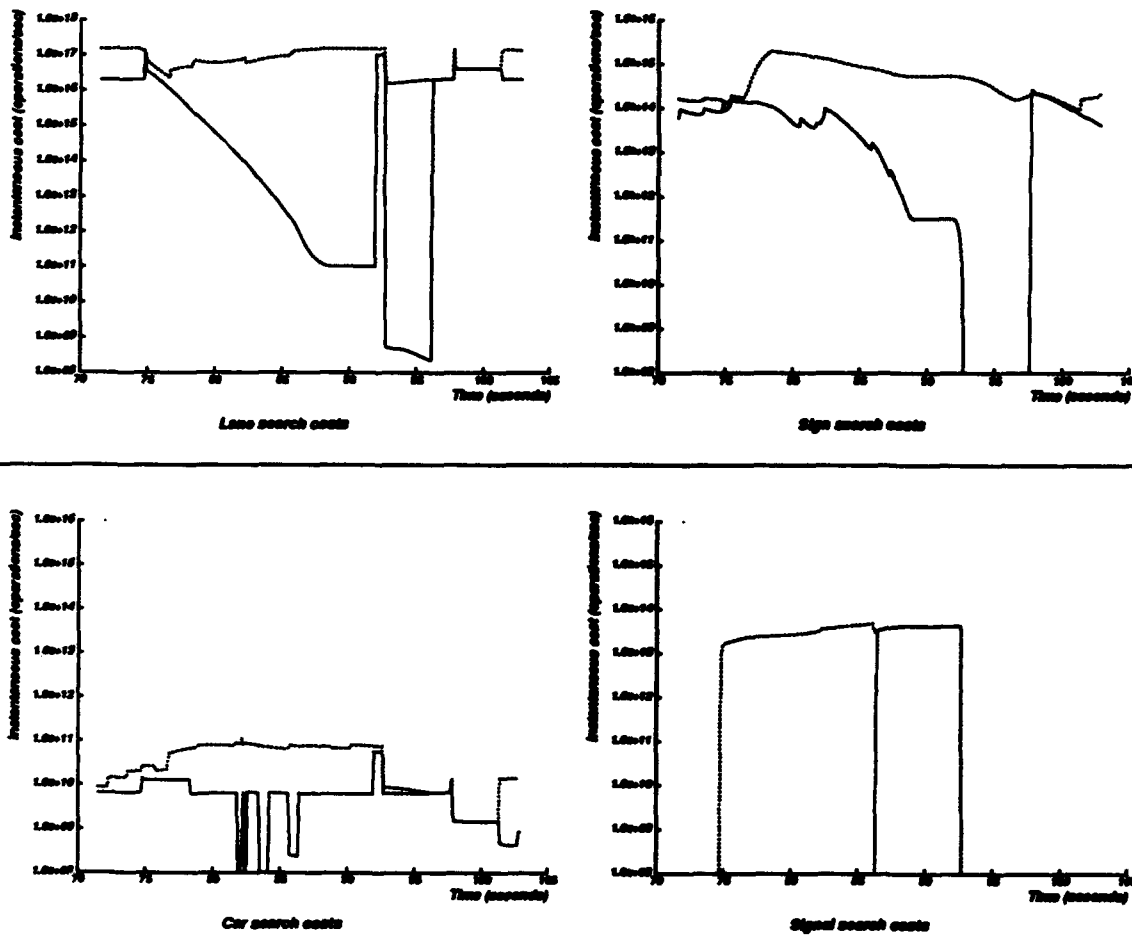


Figure 6-35: Cost of searching for various objects in Ulysses-2 (solid line), compared with Ulysses-1 (dotted line) for traffic light scenario.

6.4.5. Multiple Intersections

The final scenario involves multiple intersections, as Figure 6-36 shows. Figures 6-37 through 6-39 show how perceptual cost varies during the scenario. The costs reflect the combined effects of two intersections. When the sensors detect the intersection (point 1 of Figure 6-37), costs drop until the intersection becomes important at $t = 17.5$. Ulysses-2 looks for cross traffic and signals around the first intersection until the robot enters the intersection (point 3 in Figure 6-37). Perceptual cost is low approaching in the second intersection because Ulysses-2 finds the Stop sign and does not search the intersection further. In fact, when the robot begins maximum braking for the Stop sign at around $t = 23$, even car following can have no further impact and Ulysses-2 stops looking for cars altogether. (This is probably a bug; the worst case for car following should be emergency braking, which is more severe than nominal maximum braking. Car perception rates would then stay at around the 10^{10} level.) After the robot stops at the sign at $t = 28$, Ulysses-2 searches for cross traffic and follows each car in turn as it approaches and enters the intersection. At about $t = 37$, the traffic clears and after a final look the robot enters the intersection.

6.5. Related Work

Ulysses-2 makes use of bounds propagation and tree search to select sensing functions. While both of these mechanisms have roots in previous work, Ulysses-2 extends them and combines them in a novel way. The propagation of numerical intervals through functions has been explored extensively in "constraint propagation" systems [Davis 87]. For example, as part of a system for arithmetic reasoning, Simmons [Simmons 86] implemented the same arithmetic interval propagation functions used in Ulysses-2. He demonstrated that unusual numerical functions could be handled with the same basic ideas; Ulysses-2 also goes beyond the normal arithmetic functions with the generic and special functions (described in section 6.2.1). Ulysses-2 extends the idea of bounds propagation even further, however, by including boolean and other symbolic functions.

The tree search algorithm in Ulysses-2 can be thought of as an extension of branch-and-bound (BB) search [Lawler 66]. BB maximizes (or minimizes) the numerical value of the tree, so it effectively searches a tree of Max (or Min) nodes. BB uses bounds on the values of unexplored nodes to estimate where the solution is. Whenever a branch of the tree is explored, the best value previously found on other branches is used as a cutoff to stop search if the value of the current branch falls too low. A similar technique is also used to search game trees with *both* Min and Max nodes (e.g., B* [Berliner 79]). Ulysses-2 uses bounds and cutoffs the same way on Min and Max nodes, and extends the concept to other arithmetic and symbolic nodes as well. In addition, when leaf nodes are evaluated in Ulysses-2, they

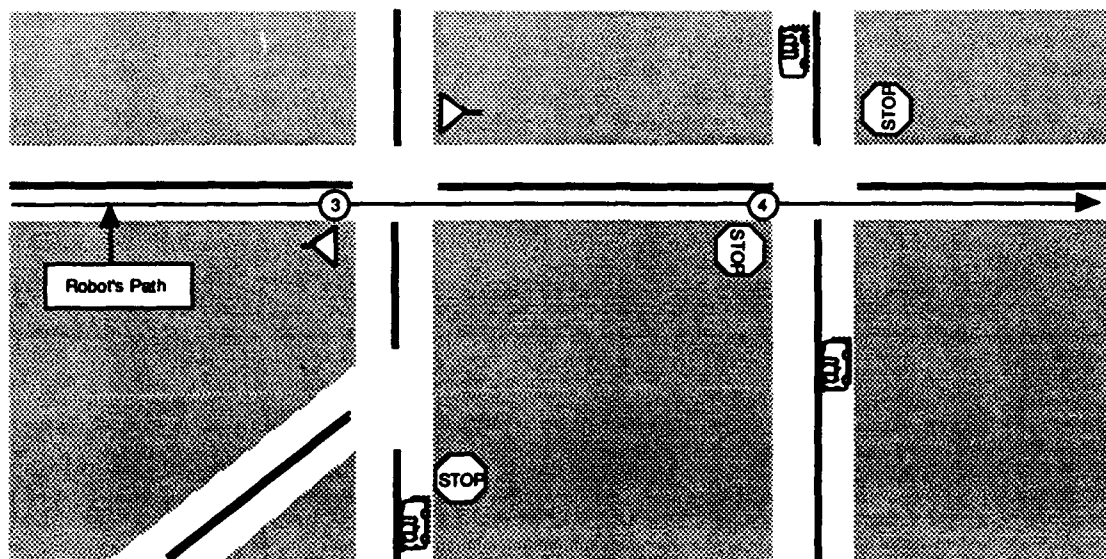


Figure 6-36: Multiple intersection scenario (not to scale).

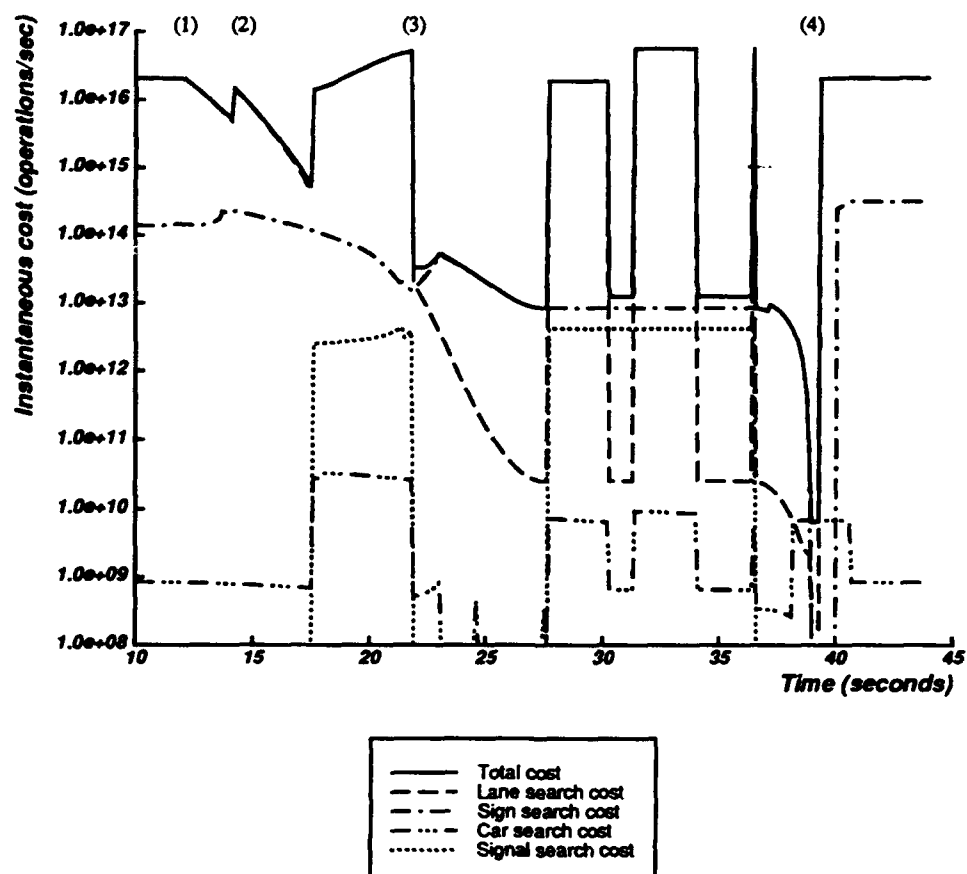


Figure 6-37:

Perceptual costs for Ulysses-2 during multiple intersection scenario. Notes correspond to figure above, except: (1) sensors reach first intersection; (2) sensors reach second intersection.

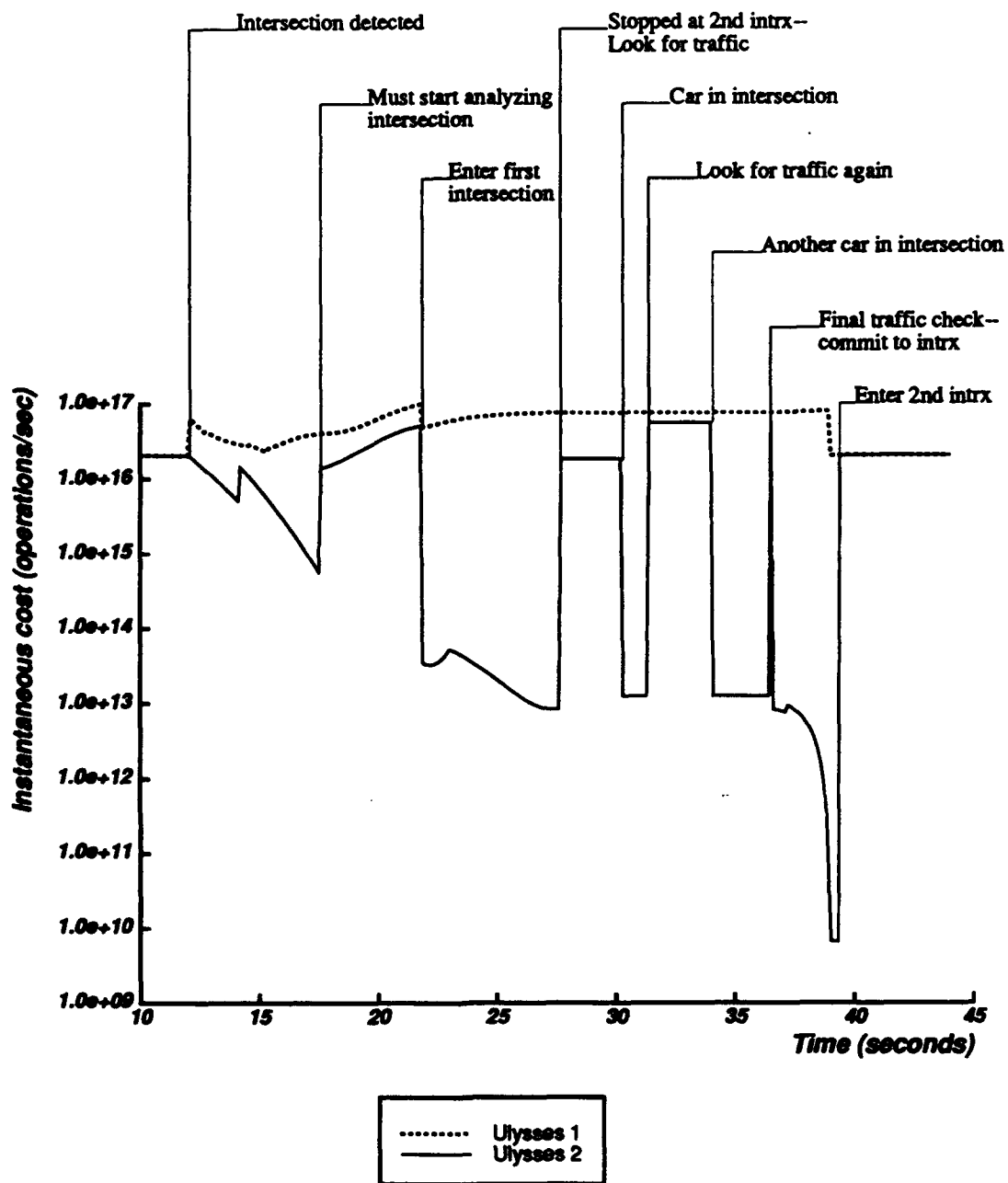


Figure 6-38: Total perceptual cost of Ulysses-2 compared to Ulysses-1 for multiple intersection scenario.

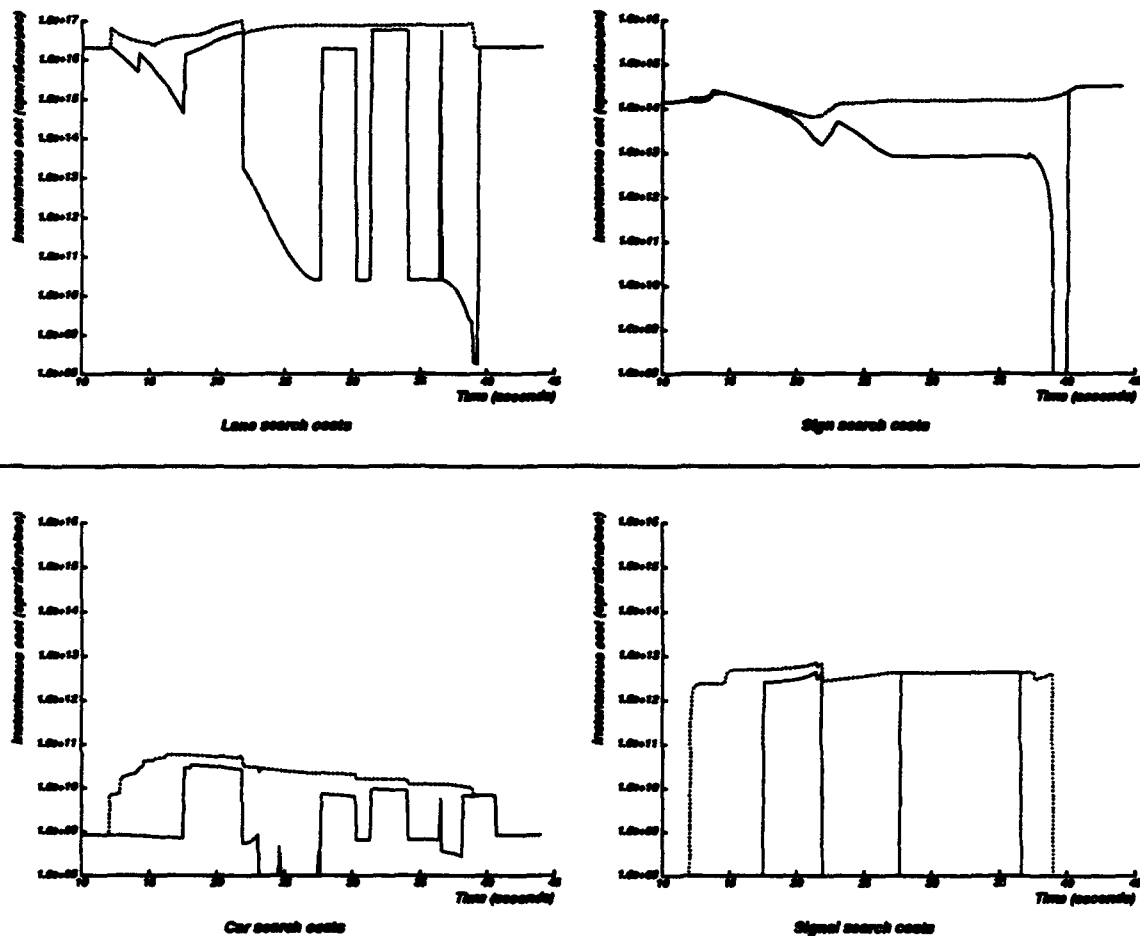


Figure 6-39: Cost of searching for various objects in Ulysses-2 (solid line), compared with Ulysses-1 (dotted line) for multiple intersection scenario.

cannot be copied back to higher levels of the tree as in BB; because of the different functions, the bounds propagation mechanism must be used.

6.6. Summary

Ulysses-2 implements the Ulysses driving model with explicit data structures for both the world model and the decision logic. The world model (corridor structure) has slots for every traffic object that the driving logic uses. Procedures for filling in this structure using perceptual routines are encoded with each slot. The driving logic is represented as an inference tree. Leaf nodes sense the world by connecting to the corridor structure. Each leaf node senses one fact, and defaults to a worst-case range of values when the fact has not yet been sensed. These default bounds are propagated up the tree to the root, where they translate into a range of possible actions. Ulysses-2 evaluates the tree by searching for the next most critical fact to sense. Perceptual routines are run, the corridor structure is updated, and the new sensed value is propagated up the tree. This process continues until the root node specifies only one action.

Although the Ulysses-2 search process is not optimal, it is nevertheless efficient. It is not optimal because it does not consider all sensing sequences to determine which is the cheapest to accomplish its goals. Neither does Ulysses-2 attempt to be optimal in a probabilistic sense, since it does not use probability distributions for node values, nor estimate likely outcomes of sensing actions. These approaches would be very costly due to the larger search space involved, yet it is doubtful that they could guarantee absolute minimum perceptual cost anyway because of the dynamic nature of the inference tree. Ulysses-2 instead just uses the extreme inputs to functions such as Max and Min to make some critical sensing choices almost for free. Ulysses-2 makes the most of this simple search algorithm by propagating every newly sensed fact as far as possible through the tree and cutting off exploration whenever a node's bounds collapse to a single value or fail to meet a cutoff. This approach is very effective because at the top of the inference trees, the driving model always includes some prioritizing knowledge—expressed in, e.g., Min and Max functions—to rank possible actions. Thus tree branches can be pruned near the top, thereby eliminating a lot of potential sensing. I conjecture that many task descriptions share this characteristic and would thus benefit similarly from this search technique.

The five scenarios show the effectiveness of Ulysses-2. In situations in which there is little choice of action, and few constraints on the robot, Ulysses-2 shows no improvement over Ulysses-1. The two-lane highway segments are examples of this. When there is a choice of lanes, as in four-lane highways, Ulysses-2 offers about an order of magnitude of improvement because it prunes away the unnecessary search of the adjacent lane. Ulysses-2 makes its biggest impact when there are several factors that could significantly constrain the

robot's actions. In all scenarios that contained an intersection, Ulysses-2 was at times able to find a constraining condition early in its search of the decision tree and prune away most of the rest of the tree. This resulted in a cost savings of from one to *six orders of magnitude* over Ulysses-1. The next chapter will describe yet further improvements in the driving system through the use of a persistent world model.

Chapter 7

Ulysses 3: Modeling World Dynamics

The first two implementations of the driving program reduce the perceptual load on the robot when it looks at the world each decision cycle. However, both Ulysses-1 and Ulysses-2 throw away all information between cycles. The third implementation of the driving program, Ulysses-3, takes advantage of coherence in the world over time to further reduce the information it needs to sense. Knowledge about how each object can change is stored with the sense node for that object, and Ulysses-3 automatically determines when the robot needs to sense that object again.

In general terms, Ulysses-3 makes the same computations as Ulysses-2 does. However, all of the sensed information is kept around from cycle to cycle; time stamps on the information record how old it is. At the beginning of each new cycle, programs in the sense nodes take the age and estimate what the new bounds on the data should be. If these new bounds do not affect the robot action, then the sense node will not be explored by the search algorithm. Thus specific knowledge about the dynamics of objects in the domain is automatically incorporated into the selective perception process.

7.1. Data Structure Modifications

Ulysses-3 adds two features to the data structures defined for Ulysses-2. First, in the corridor structure, data items incorporate a new field. This field records the time when the data is sensed. The corridor structure thus becomes a time-stamped data base. The status field of data items also allows a new stat, "old," for the data item. Where "new" means not-yet-sensed, and "current" means just-sensed, "old" means that the data was sensed in a previous decision cycle.

The inference tree definition is also different in Ulysses-3. The sense node initialization programs do not always generate the same initial node value bounds, but instead examine the data item's time stamp and any other available information to compute the appropriate initial bounds for the current decision cycle. As with the default initial bounds of Ulysses-2, "appropriate" computed bounds must be worst-case values so that sensing can only yield a value between the bounds. Some objects can be fixed with high confidence, while others may

change unpredictably between cycles. For example, the *type* of a sign does not change over time. The *range* to the sign is updated according to the motion of the robot since the last decision cycle, which is assumed to be known⁶. Ulysses-3 estimates new states for cars on cross streets by hypothesizing high and low acceleration rates and computing the possible bounds on speed and position. An example of an unpredictable object is the car ahead of the robot. It is presumably possible for a car to change lanes and move in front of the robot at any time. The driving module does not yet include an analysis of traffic patterns on multiple lanes in front of the driver; therefore Ulysses-3 cannot predict when such a lane change could occur, and must find the lead car from scratch every decision cycle.

7.2. Algorithm Modifications

There are two differences between the Ulysses-3 and Ulysses-2 inference procedures. First, instead of creating the corridor structure and inference trees from scratch each decision cycle, Ulysses-3 *ages* the existing corridor and *re-initializes* the inference trees. Aging requires changing the status of all "current" data items to "old." In addition, the system clock is incremented so that the existing time stamps move farther into the past. Ulysses-3 re-initializes an inference tree by running the sense node initialization programs to compute new bounds for the data values, and then propagating the new bounds back up the tree. Figure 7-1 illustrates this process. After the corridor and inference tree have been updated, Ulysses-3 uses the same search algorithm as Ulysses-2 to select sensing actions.

The second difference between the implementations is that Ulysses-3 must modify the corridor and inference trees whenever the robot moves to a new part of the corridor. For example, when the robot enters an intersection, the road part that it just left is discarded and the Start Info (see Figure 6-3) is connected to the intersection part. Similarly, the part of the inference tree that is between the root node and the old corridor part is discarded. Changing lanes requires similar changes, but between adjacent lane parts.

7.3. Experimental Results

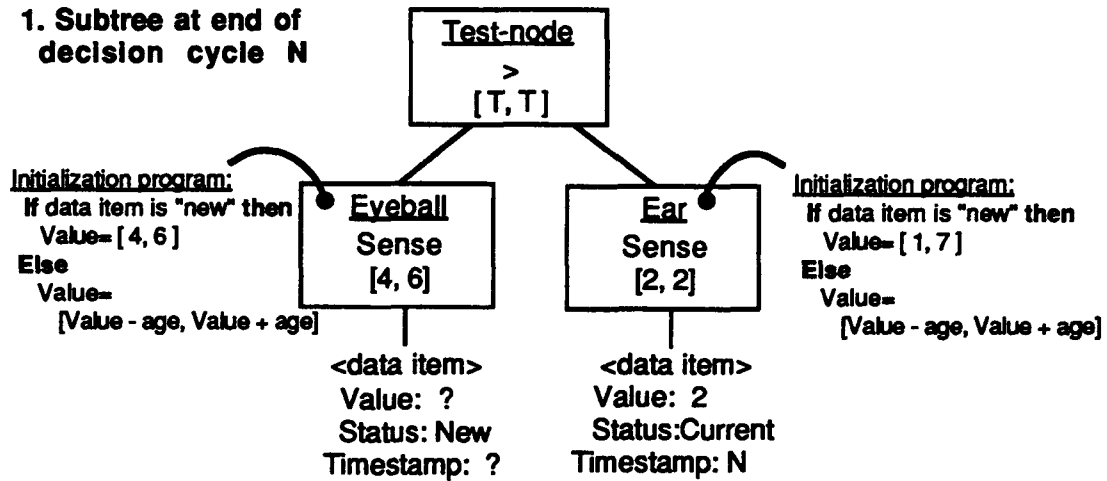
Ulysses-3 was used to drive the simulated robot through the same scenarios that were used for Ulysses-1 and Ulysses-2. Ulysses-3 produced the exact same acceleration and lane commands as the previous implementations. This section shows the estimated perceptual cost of driving the robot through the scenarios.

The estimated perceptual cost reaches a minimum of about 1.3×10^8 operations per second

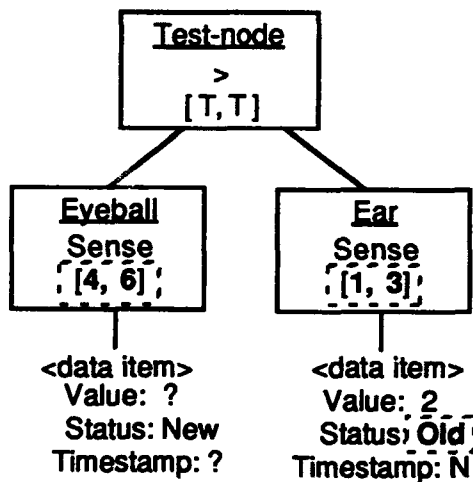
⁶If robot motion is not completely certain, then the range to signs and other similar values would have spreading bounds.

Ulysses-3 Tree Initialization

1. Subtree at end of decision cycle N



2. Cycle N+1: Age corridor and Re-Initialize sense nodes



3. Propagate new bounds up

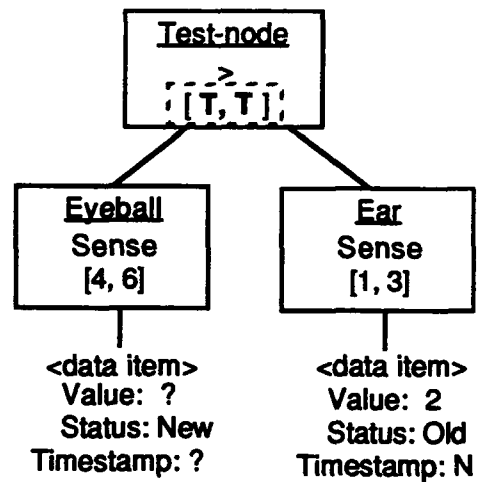


Figure 7-1:

Initialization of Ulysses-3 inference tree. The data items in the corridor are first aged. Sense nodes are then re-initialized by programs in the nodes; note that "age" = cycle# - timestamp. Finally, updated sense node values are propagated up the tree. In this case the value of Test-node does not change, so no search is necessary to evaluate the tree.

in several scenarios. This is the cost of finding the lane in front the robot (see Appendix B). All implementations of Ulysses currently assume that this is a basic reference action, and it is not even included in the inference tree in Ulysses-2 or Ulysses-3. Thus this action cannot be eliminated, and the associated cost is the absolute minimum that Ulysses-3 can achieve. Future implemenations may include the lane-finding action in the reasoning process, thereby allowing the robot to drive completely blind (tactically) in some situations.

7.3.1. Left Side Road

The first scenario is again the left side road, shown in Figure 7-2. Figure 7-3 shows the perceptual cost of using Ulysses-3, including a breakdown by object type. Clearly, the character of the cost profile is much different than for the previous two implementations. Figure 7-4 shows a direct comparison with the Ulysses-2 costs. This figure shows that most of the time, the cost for Ulysses-3 is five to six orders of magnitude less than for Ulysses-2. At intervals of several seconds, there are sudden spikes in the cost profile that reach all of the way up to the Ulysses-2 levels. This graph reflects the fact that Ulysses-3 remembers where some objects are from moment to moment, and only requests new information when it becomes too uncertain about the world state.

Figure 7-5 shows more specifically what Ulysses-3 is doing. Before the intersection is detected at $t = 37.5$, the lane search cost is between 10^{11} and 10^{12} . This is much less than Ulysses-2 (10^{16} or so) because Ulysses-3 is looking only for new pieces of road that it could not see in the last decision cycle. The sign search cost graph in Figure 7-5 has sharp spikes separated by periods of zero cost. The spikes occur when Ulysses-3 searches along the road ahead for the next sign. If this next sign is far ahead, then Ulysses-3 is able to wait some time before extending its sign horizon again.

When the intersection is detected there is one decision cycle of high lane, sign, and car search cost as Ulysses-3 considers changing lanes. In this scenario it is in fact necessary for the robot to change lanes in preparation for making a left turn. Cost drops during the lane change because the Ulysses-3 temporarily stops checking for an intersection in the distance. (See also the discussion of two-lane vs. four-lane lane search, below). There is another spike at about $t = 41.5$ when the robot finishes the lane change as Ulysses-3 makes sure there is no lane farther to the left. Otherwise, Ulysses-3 does not have to look for lanes or signs or signals again until the intersection becomes important at $t = 43$. At this point there is a sharp increase in cost because Ulysses-3 looks for many objects while it analyzes the intersection. After the rise, the cost drops again for several seconds. This indicates that although Ulysses-3 has found the approaching car, it does not keep looking at it because it can estimate when the car could enter the intersection. Ulysses-3 does not look again until about $t = 46$. Figure 7-6 illustrates how Ulysses-3 makes this estimate. Note also that Ulysses-3 looks only *once* for signs and signals at the intersection.

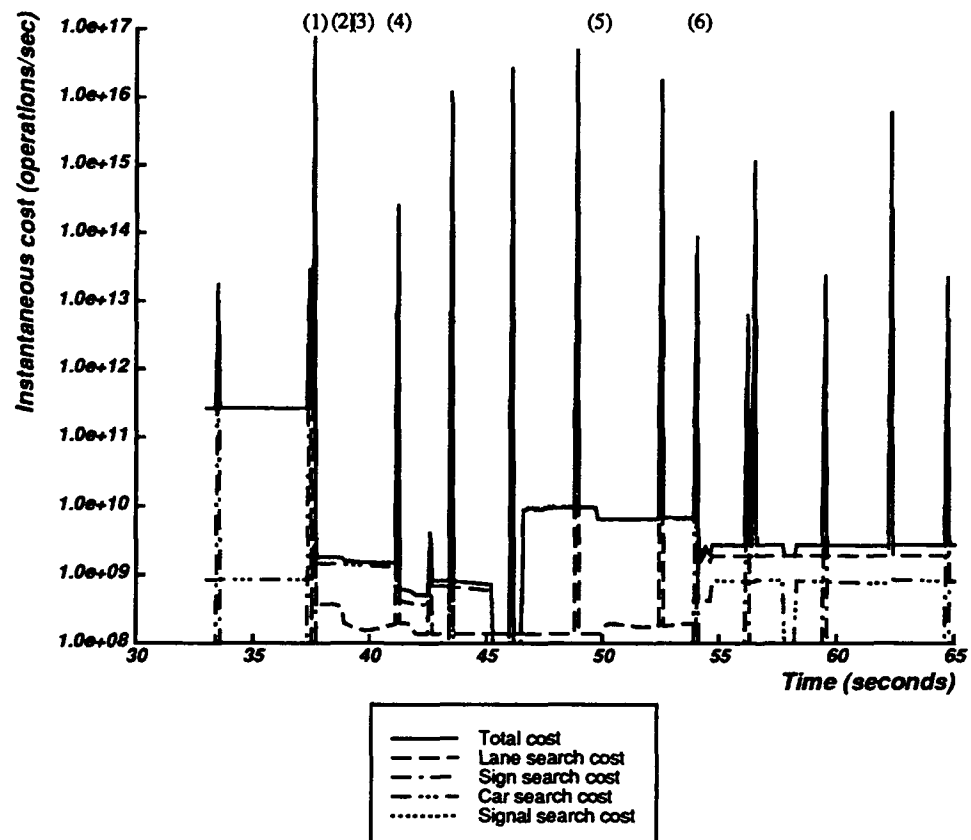
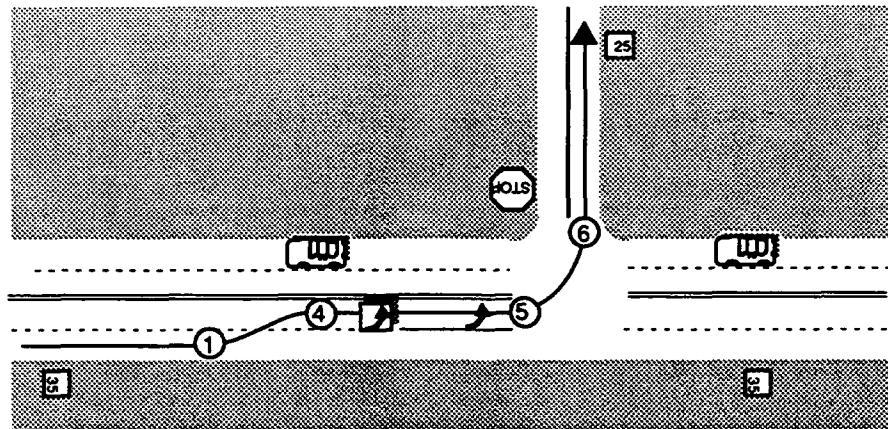


Figure 7-3: Perceptual costs for Ulysses-3, left side-road scenario. Notes correspond to figure above, except: (1) sensors reach intersection; (2) sensors reach robot's street on far side of intersection; (3) sensors reach all intersection approach roads.

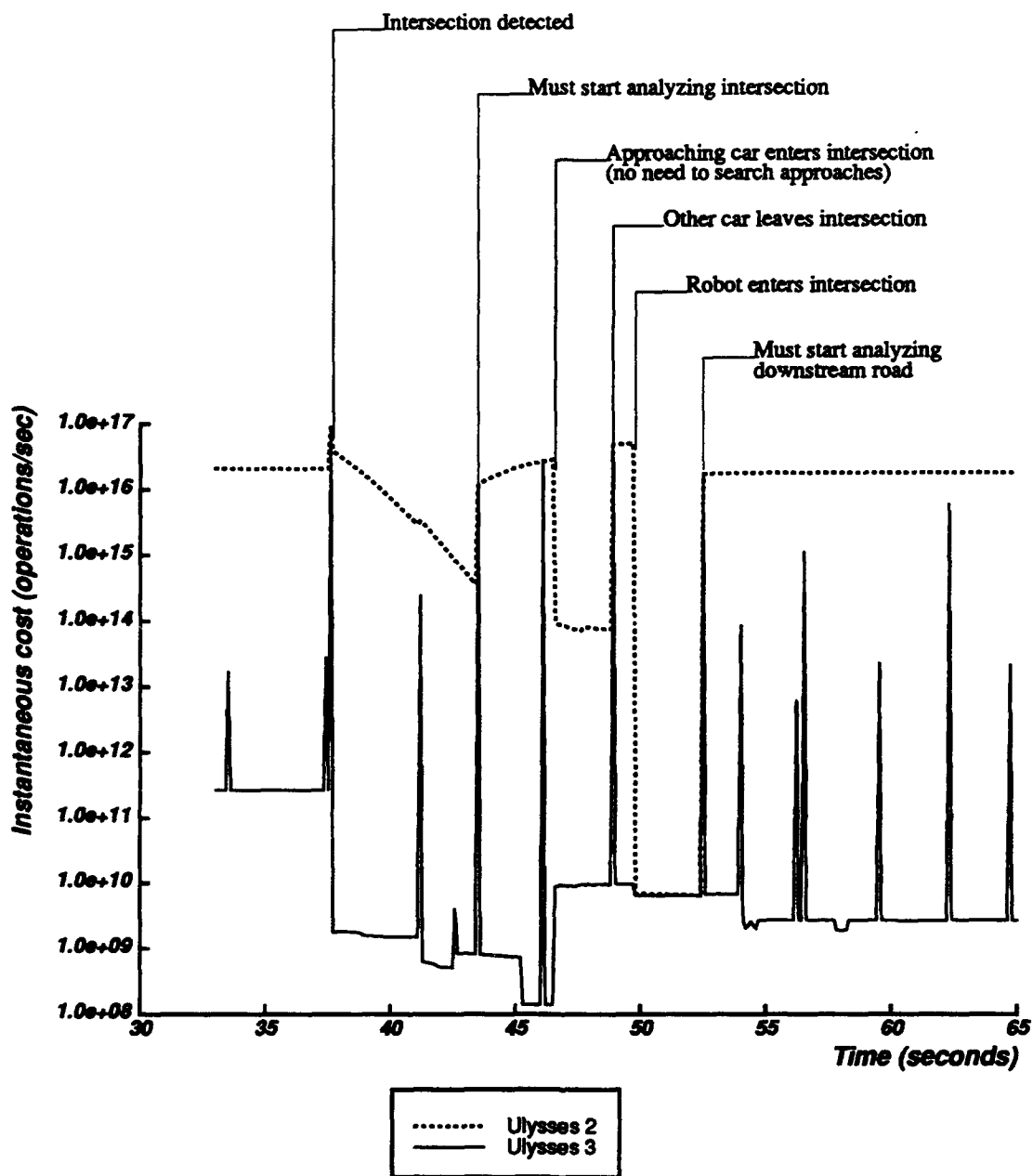


Figure 7-4: Total perceptual cost of Ulysses-3 compared to Ulysses-2 for left side-road scenario.

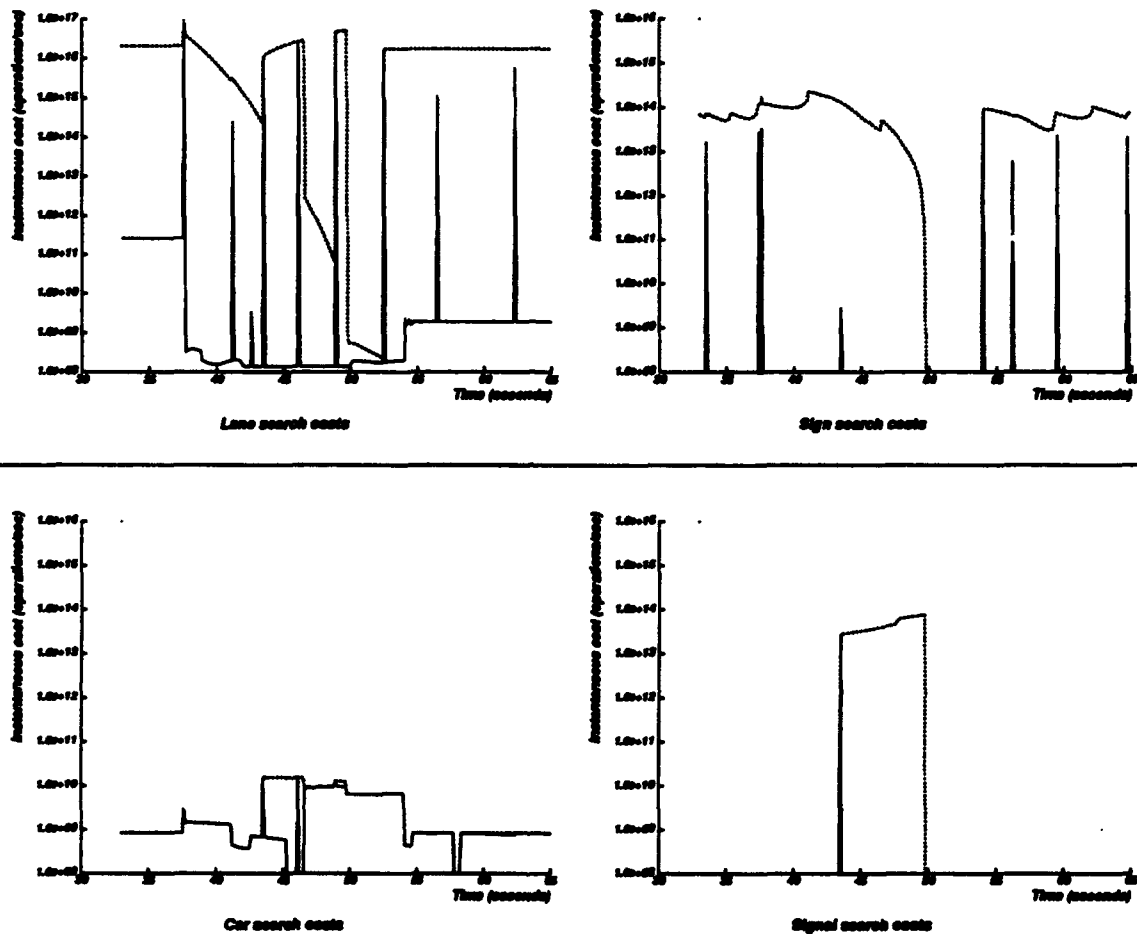


Figure 7-5: Cost of searching for various objects in Ulysses-3 (solid line), compared with Ulysses-2 (dotted line) for left side-road scenario.

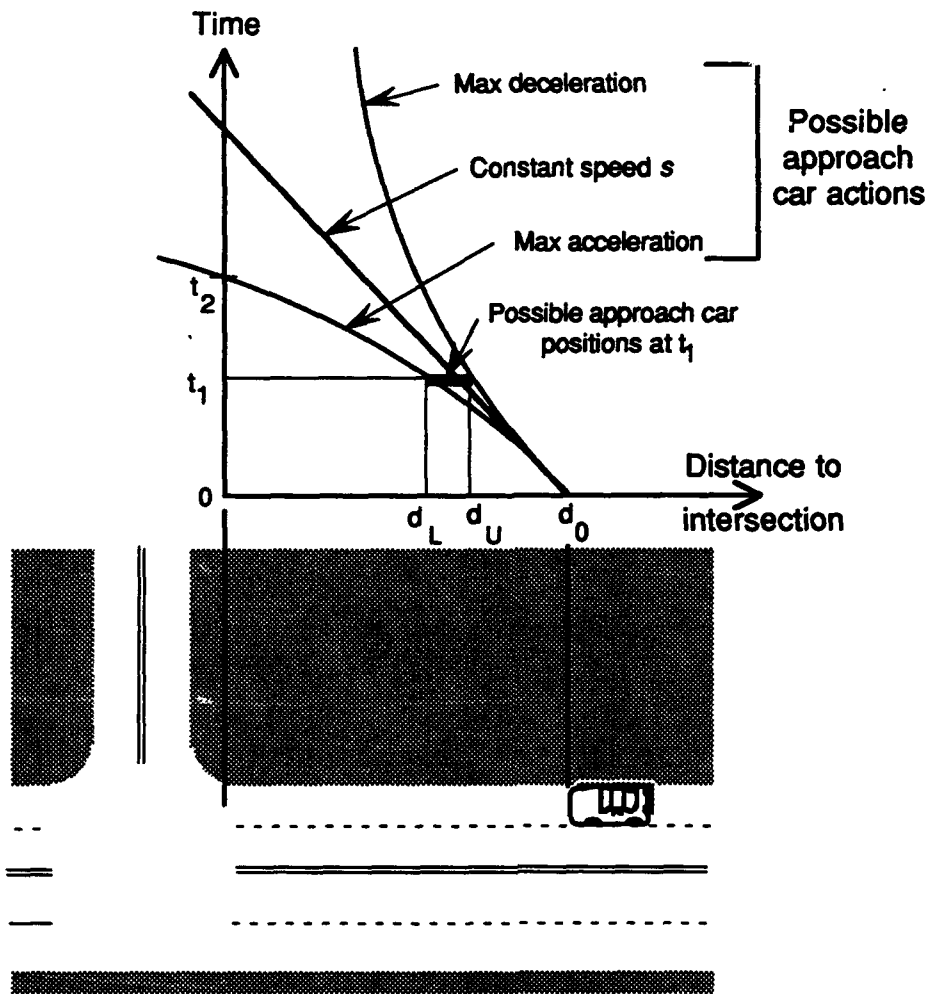


Figure 7-6:

Estimating the arrival time of an approaching car at the intersection. At $t = 0$, the car is observed at distance d_0 , travelling at speed s . At time t_1 later, the car could be anywhere from d_L to d_U away from the intersection. t_2 is the earliest time that the robot should look at the approach road again to see if it is clear of cars.

When the approaching car actually does enter the intersection, the perceptual cost goes up a bit as Ulysses-3 watches the car. Ulysses-3 must look at the car every cycle because the perceptual routines do not provide enough information for Ulysses-3 to predict when the car could leave the intersection. The robot *could* just ignore the car for a few seconds before looking again; in this case, the car might clear the intersection some time before the robot looked again. However, in order to give Ulysses-3 the same performance as previous implementations, and allow it to produce the exact same acceleration commands, Ulysses-3 was required to detect a clear intersection as soon as possible.

When the car finally clears the intersection, Ulysses-3 makes one last check for approaching traffic before deciding to go through the intersection. This check is reflected in the spike in cost at $t = 48.5$. Perceptual cost stays low until the downstream lane becomes critical at about $t = 54$. When the robot actually enters the downstream lane, lane search costs stay lower than they were on the four-lane road, but have high-cost spikes. The two-lane road is cheaper because the robot does not have to consider changing lanes. If the robot eventually needs to change lanes, Ulysses-3 tries to find the intersection as soon as possible. Otherwise, Ulysses-3 finds a length of road ahead and doesn't look again until the road end would cause deceleration.

7.3.2. Unordered Intersection

Figures 7-7 through 7-10 show the perceptual cost of the unordered intersection scenario. The cost profile shows many of the same characteristics as the left side road scenario. The cost for finding the road and signs is low and jumps to high values periodically. Near the intersection, car search costs spike occasionally to reduce the uncertainty about the position of the approaching car. Ulysses-3 looks for signs and signals at the intersection only once.

7.3.3. Four-lane Highway

Figures 7-11 through 7-14 describe the perceptual costs of the four-lane highway scenario. As Figure 7-12 indicates, the cost profile is similar to the four-lane portion of the left side road scenario. Ulysses-3 continually looks to extend the known road ahead and periodically extends its knowledge of signs. When the program decides to pass the slower car at $t = 64$, it suddenly uses more perception as it searches the adjacent lane. Cost drops during the lane change as before. The big spike after the lane change comes when the Ulysses-3 again extends its search of the lane out to the range limit of the sensors. When the robot pulls ahead of the other car at $t = 73.5$, it searches the right lane until the program allows the lane change at $t = 75.5$.

7.3.4. Intersection With Traffic Lights

Figures 7-15 through 7-18 show the results of the traffic light scenario. The cost profile during the approach to the intersection is similar to the profile at the beginning of the left side road scenario. During the time the robot is moving into the turn lane, $t < 87$, the robot is constrained by channelization (i.e., it isn't allowed to enter the intersection until it is in the turn lane) so Ulysses-3 does not analyze the intersection. In fact, the acceleration is sometimes low enough to obviate the need to look for cars ahead in the lane (similar to the situation in section 6.4.5). When the robot is finally in the correct lane, Ulysses-3 begins looking for intersection objects, and finds a red light. Since the light constrains the robot,

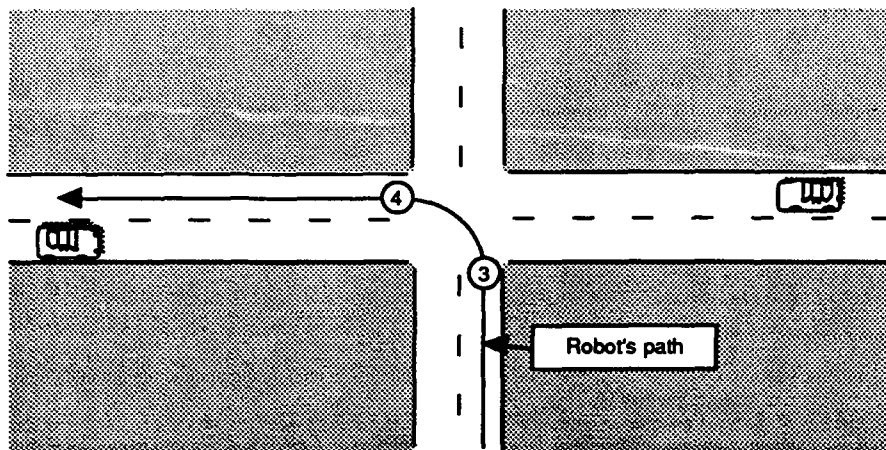


Figure 7-7: Unordered intersection scenario (not to scale).

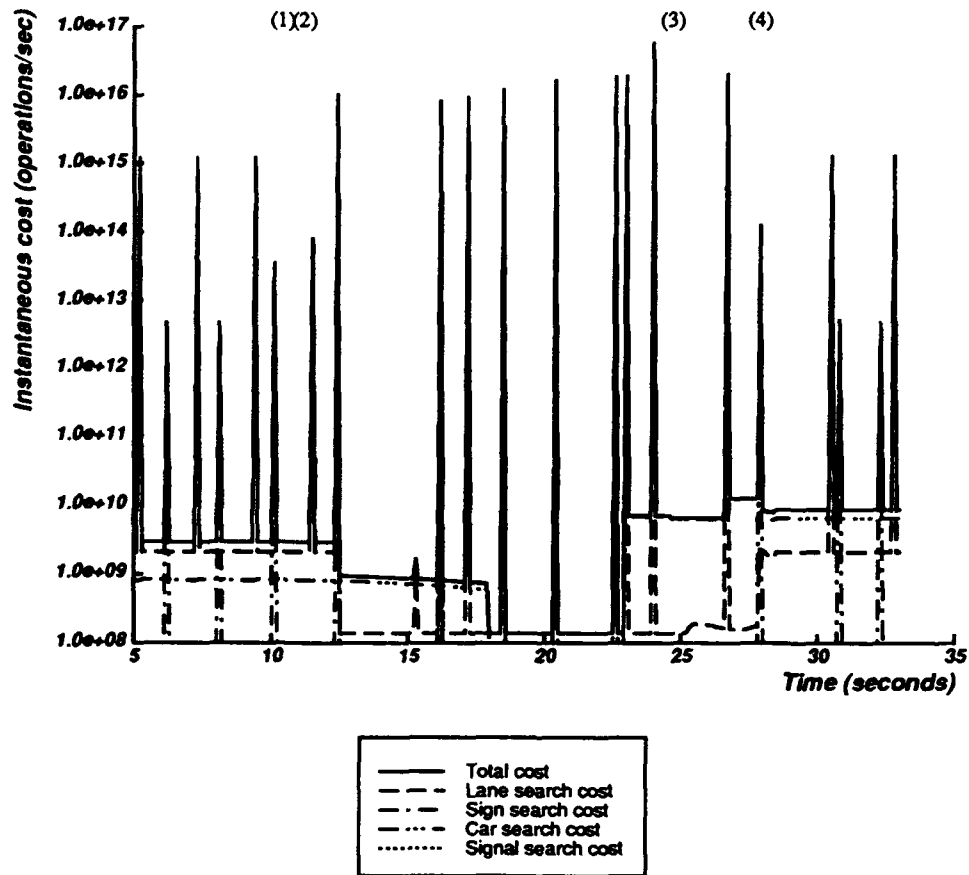


Figure 7-8: Perceptual costs for Ulysses-3 during unordered intersection scenario.
Notes correspond to figure above, except: (1) sensors reach intersection;
(2) sensors reach all intersection approach roads.

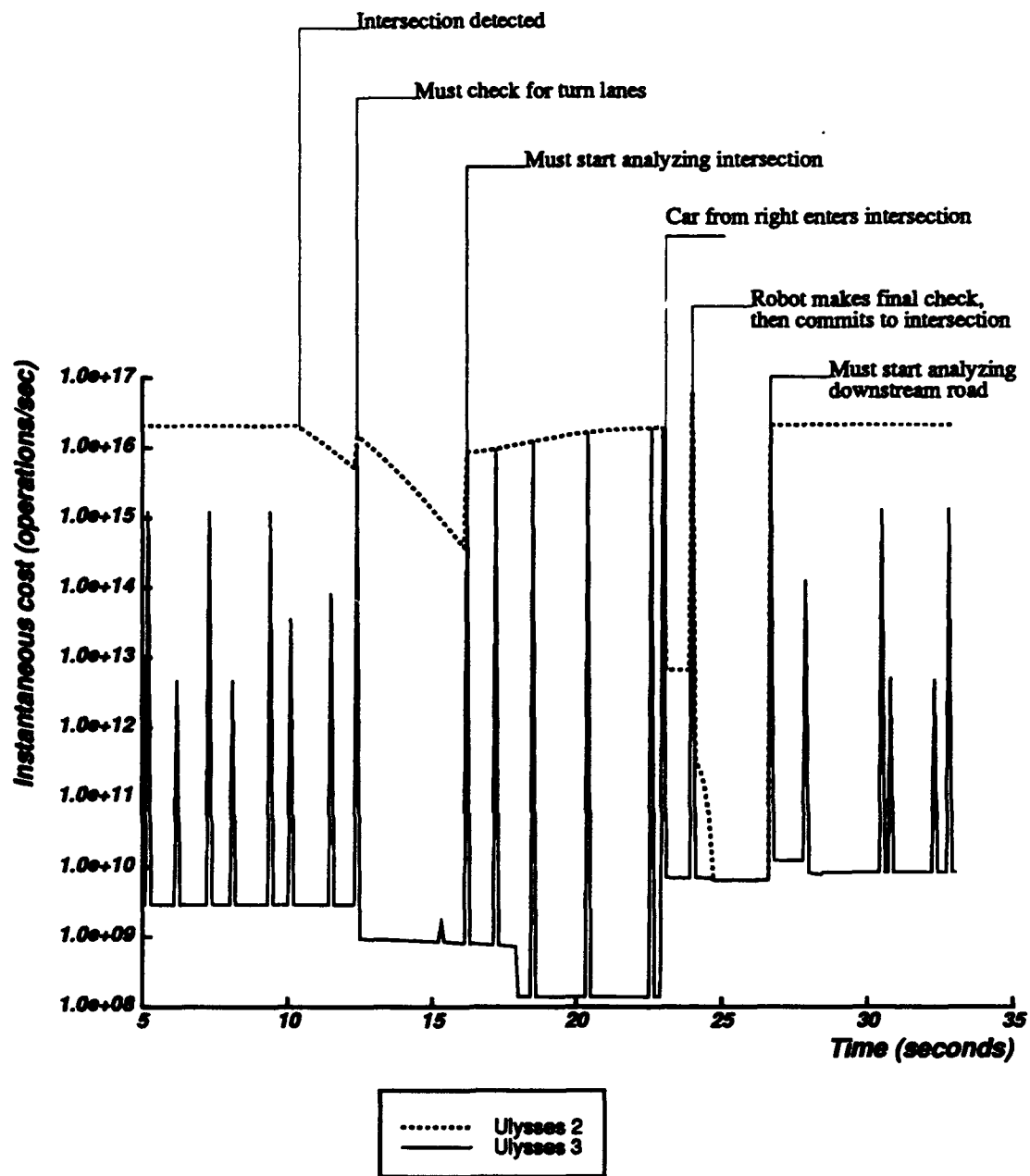


Figure 7-9: Total perceptual cost of Ulysses-3 compared to Ulysses-2 for unordered intersection scenario.

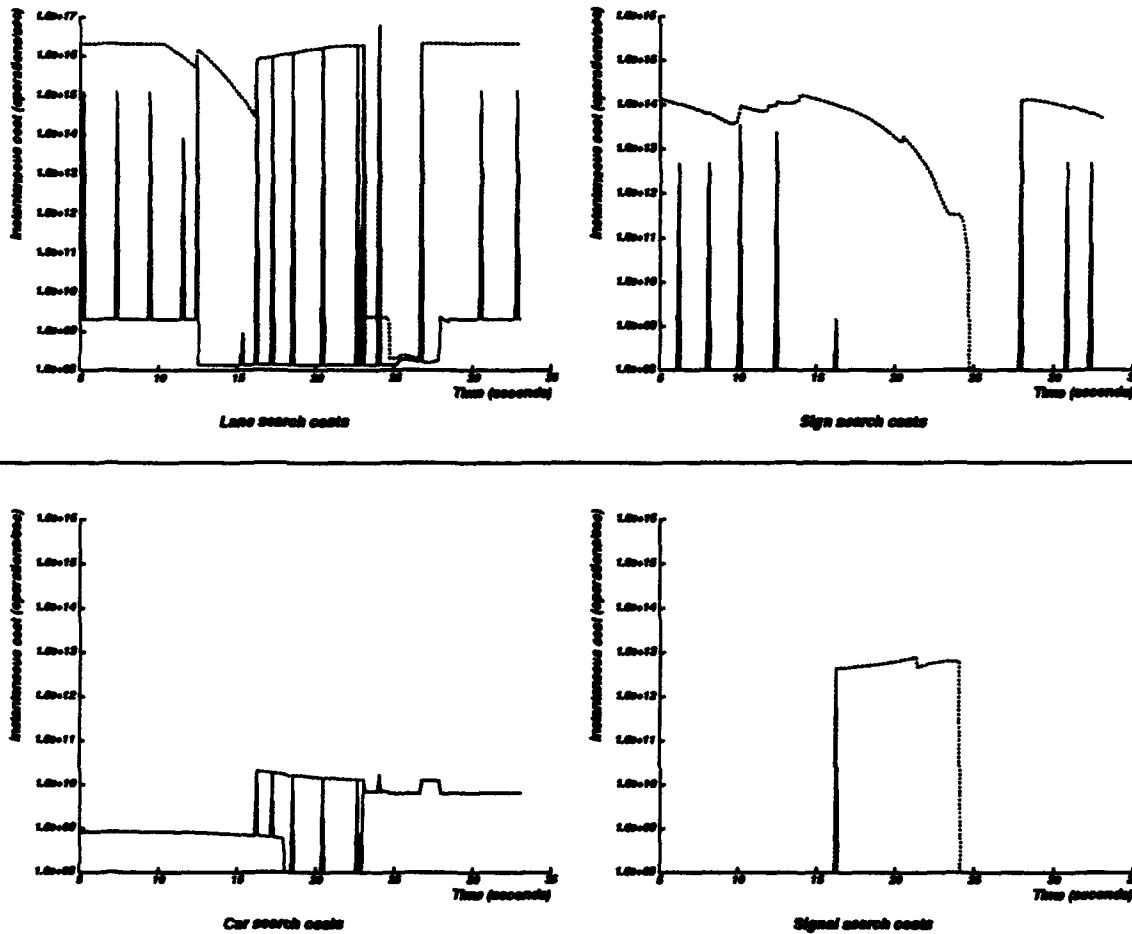


Figure 7-10: Cost of searching for various objects in Ulysses-3 (solid line), compared with Ulysses-2 (dotted line) for unordered intersection scenario.

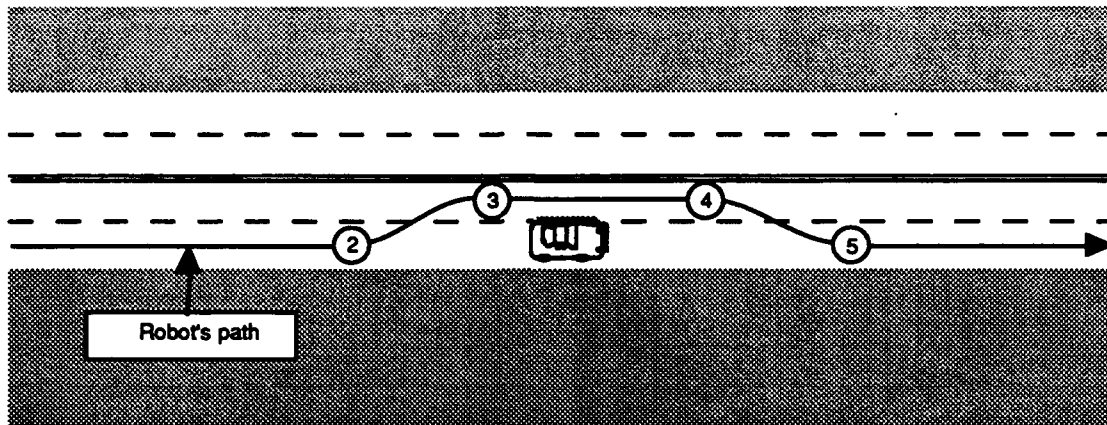


Figure 7-11: 4-lane passing scenario (not to scale).

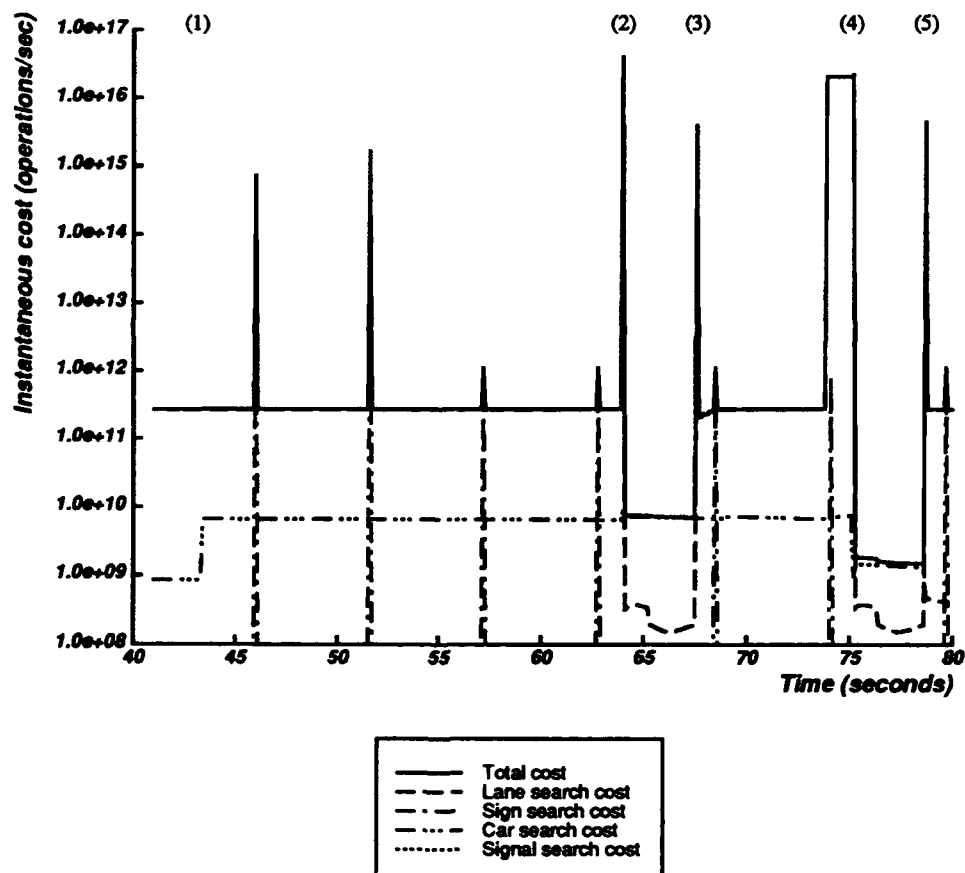


Figure 7-12:

Perceptual costs for Ulysses-3 during passing scenario. Notes correspond to figure above, except: (1) sensors reach lead car.

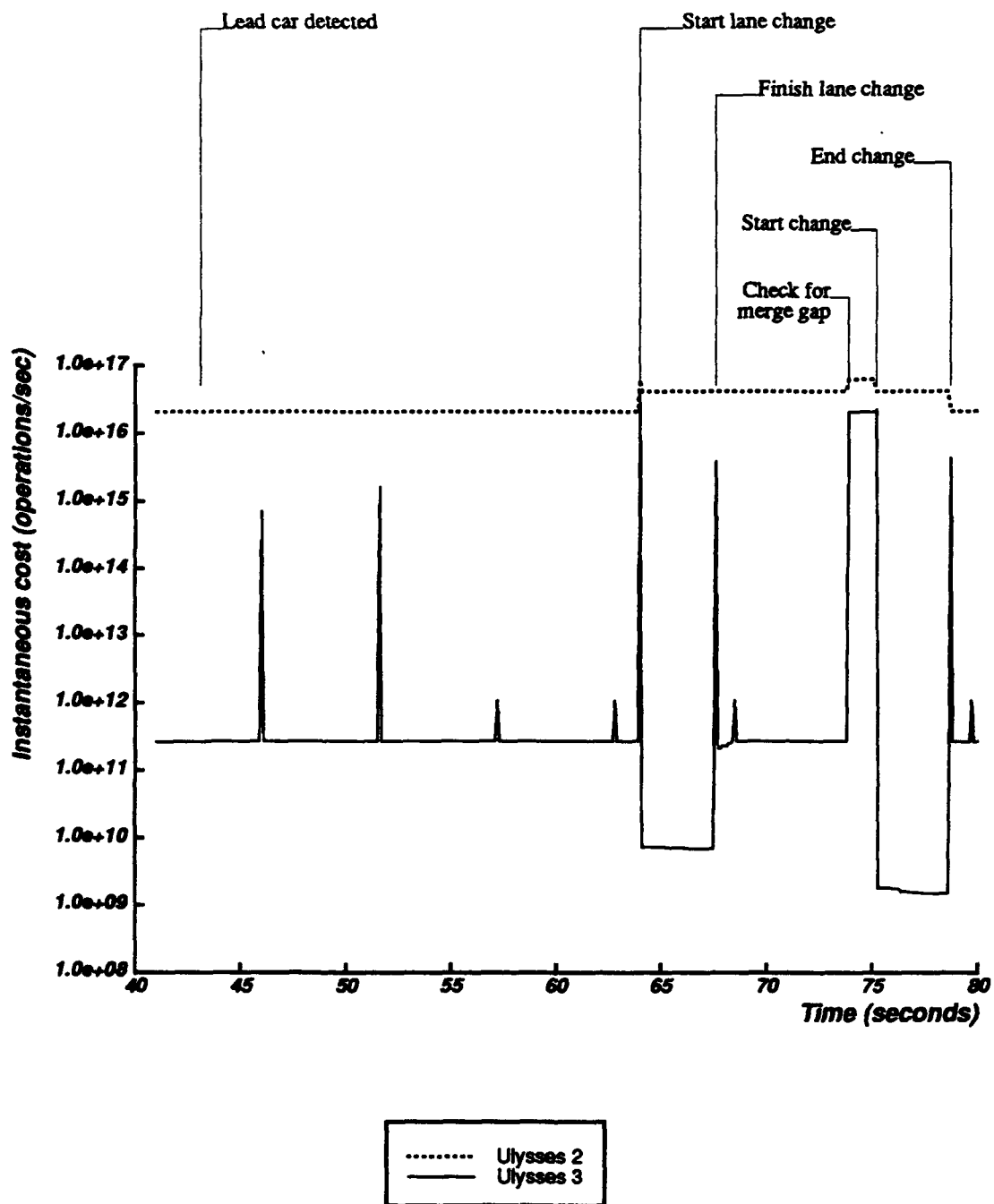


Figure 7-13: Total perceptual cost of Ulysses-3 compared to Ulysses-2 for passing scenario.

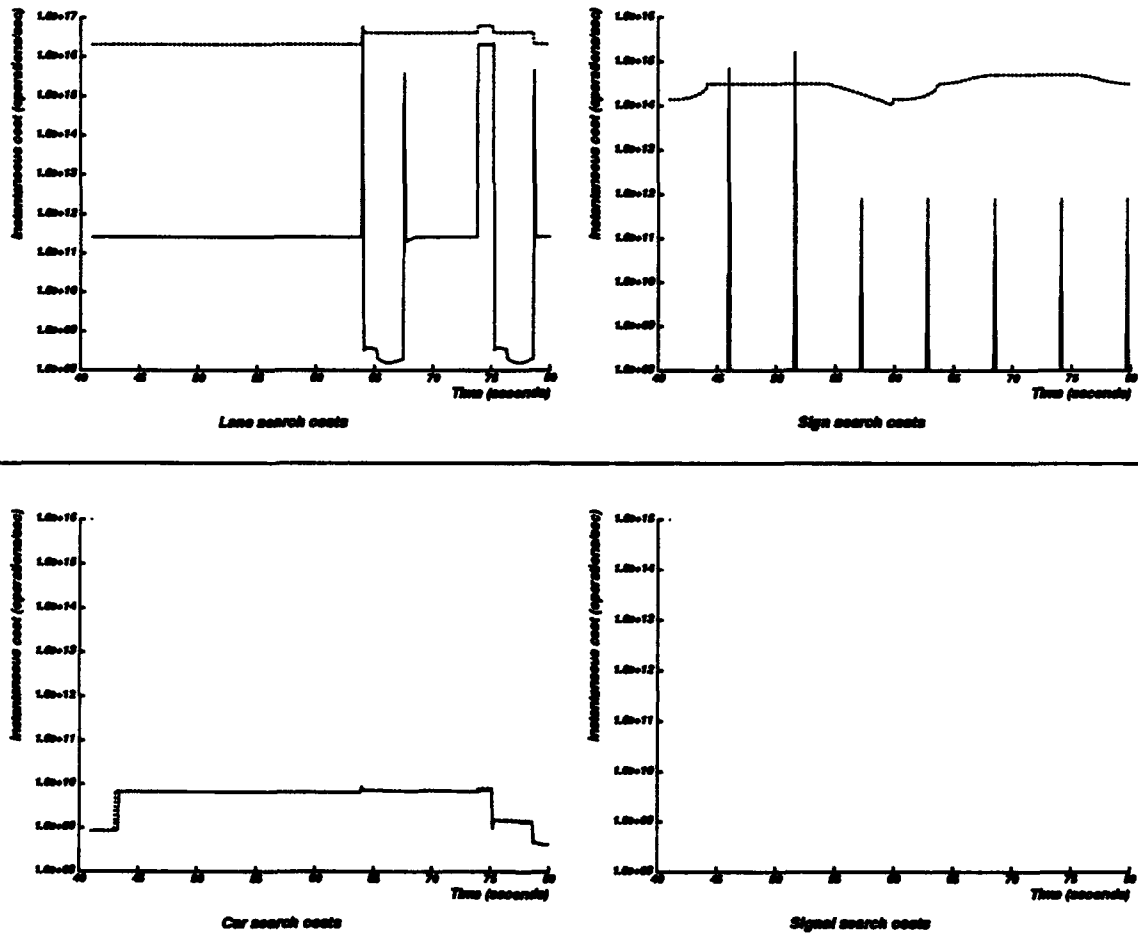


Figure 7-14: Cost of searching for various objects in Ulysses-3 (solid line), compared with Ulysses-2 (dotted line) for passing scenario.

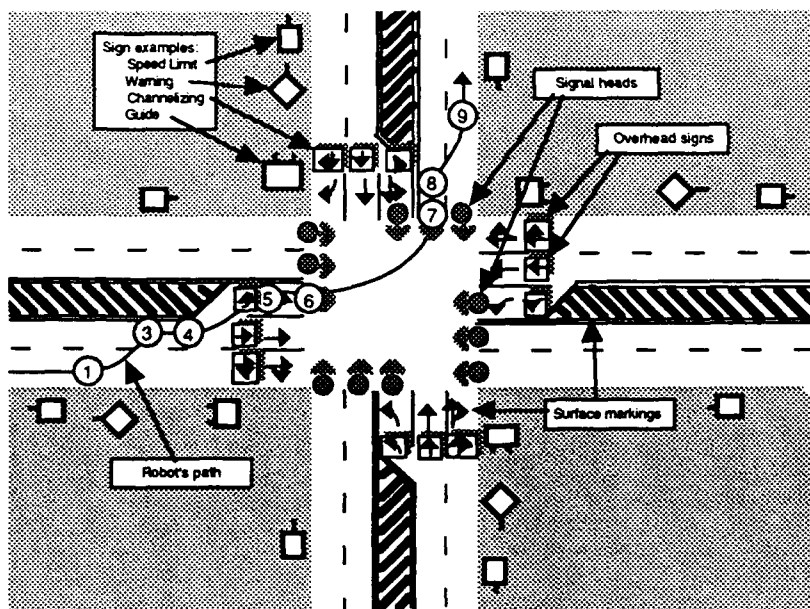


Figure 7-15: Traffic light scenario (not to scale).

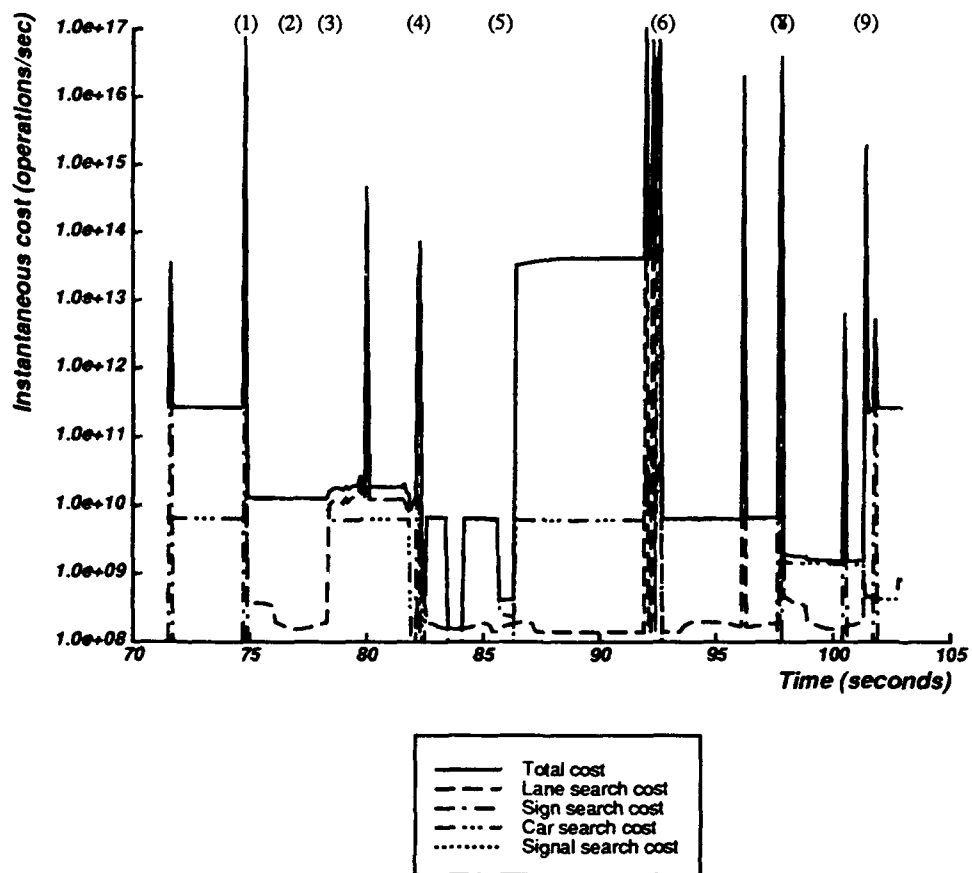


Figure 7-16:

Perceptual costs for Ulysses-3 during traffic light scenario. Notes correspond to figure above, except: (1) sensors reach intersection; (2) sensors reach all intersection approach roads.

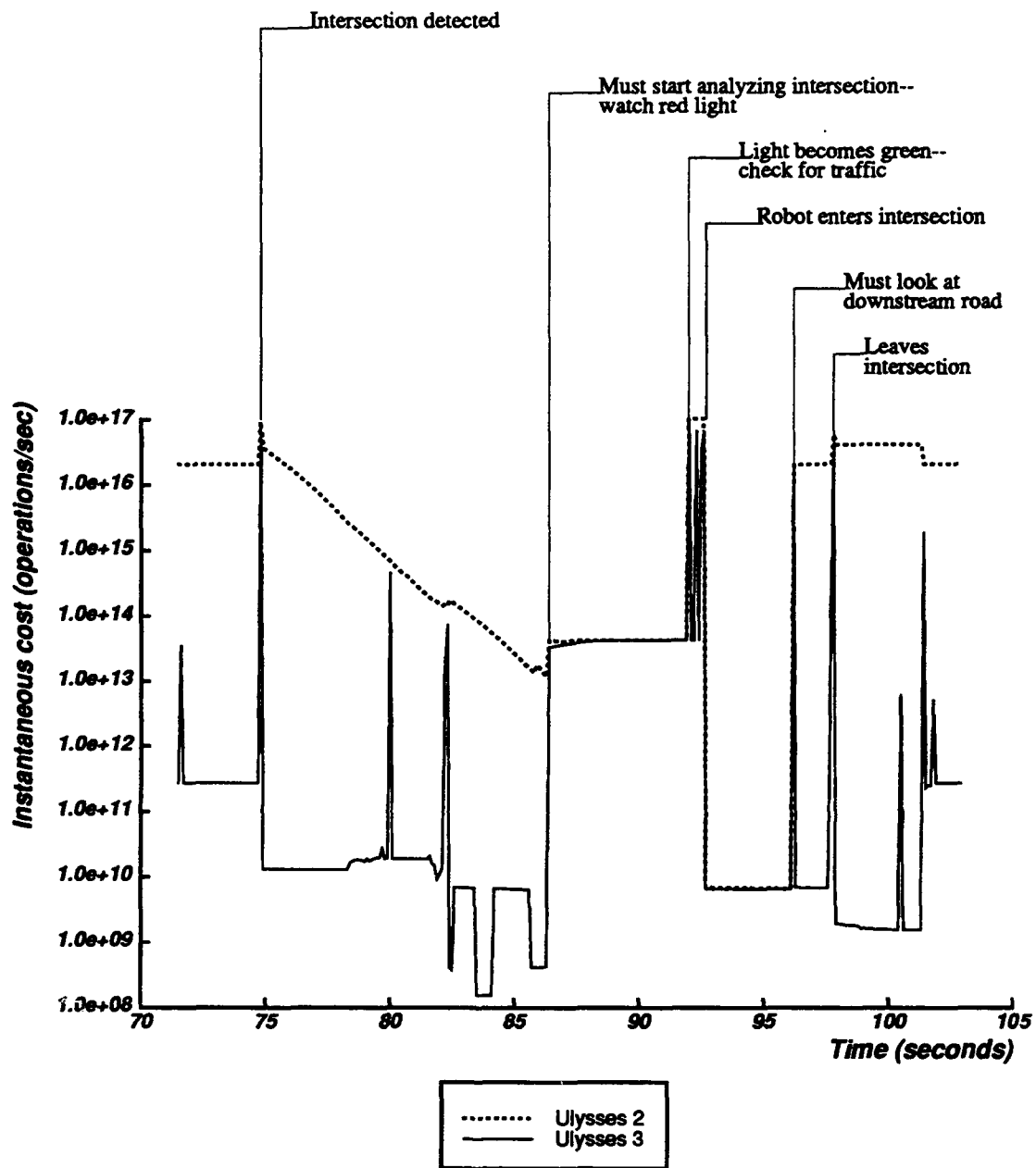


Figure 7-17: Total perceptual cost of Ulysses-3 compared to Ulysses-2 for traffic light scenario.

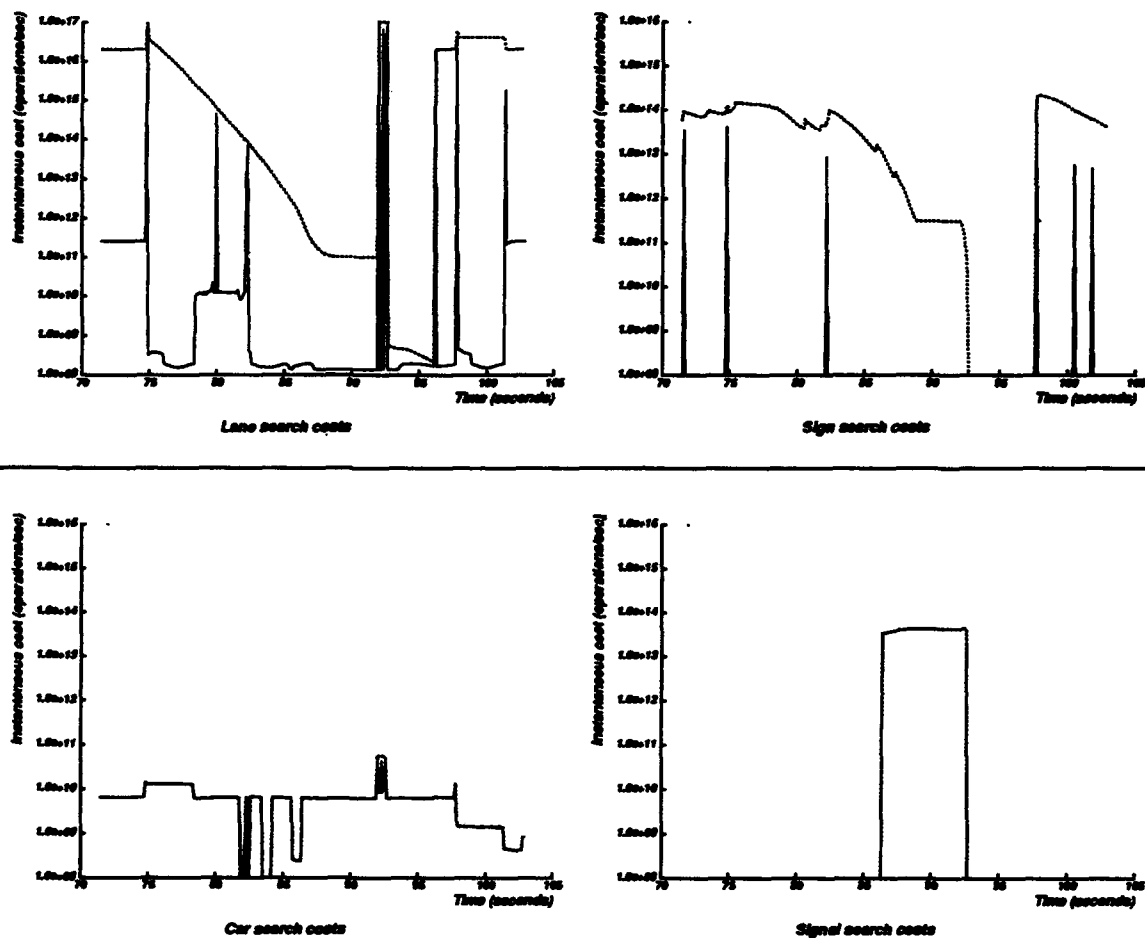


Figure 7-18: Cost of searching for various objects in Ulysses-3 (solid line), compared with Ulysses-2 (dotted line) for traffic light scenario.

Ulysses-3 does not look for traffic. However, the robot does look at the light every cycle. While it may not be crucial to react to a light immediately, Ulysses-3 is required to do this to match the performance of the other implementations.

The light eventually turns green at about $t = 92$. Ulysses-3 allows the robot to proceed, but checks the cars stopped at the light (there are some, but they are not pictured in Figure 7-15). While Ulysses-2 checks these cars every cycle, Ulysses-3 notes that they are stopped and only checks them a few times to see if they have moved. The remainder of the scenario unfolds in a manner similar to the left side road scenario.

7.3.5. Multiple Intersections

The multiple intersection scenario follows a pattern similar to previous scenarios. Figures 7-19 through 7-22 show the perceptual costs. As before, cost drops to the minimum when the robot does not have to look for the road beyond the next intersection and when signs and signals have already been found. Ulysses-3 again looks for cross cars only a few times, depending in the interim on extreme motion estimates. The net result is that the robot looks at very little most of the time.

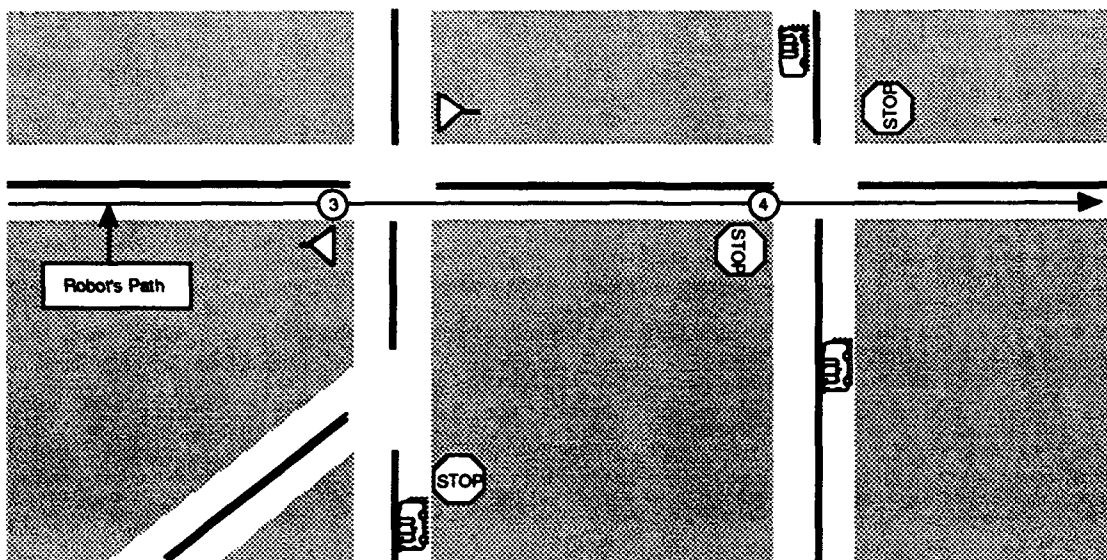


Figure 7-19: Multiple intersection scenario (not to scale).

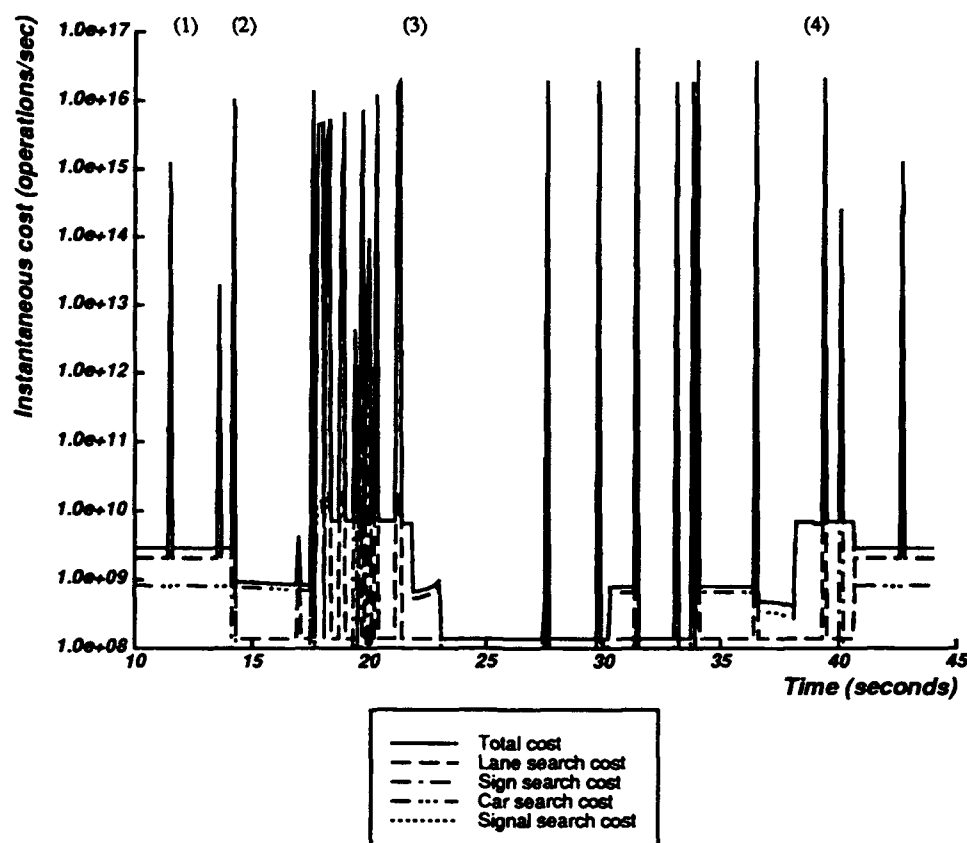


Figure 7-20: Perceptual costs for Ulysses-3 during multiple intersection scenario.
Notes correspond to figure above, except: (1) sensors reach first intersection;
(2) sensors reach second intersection.

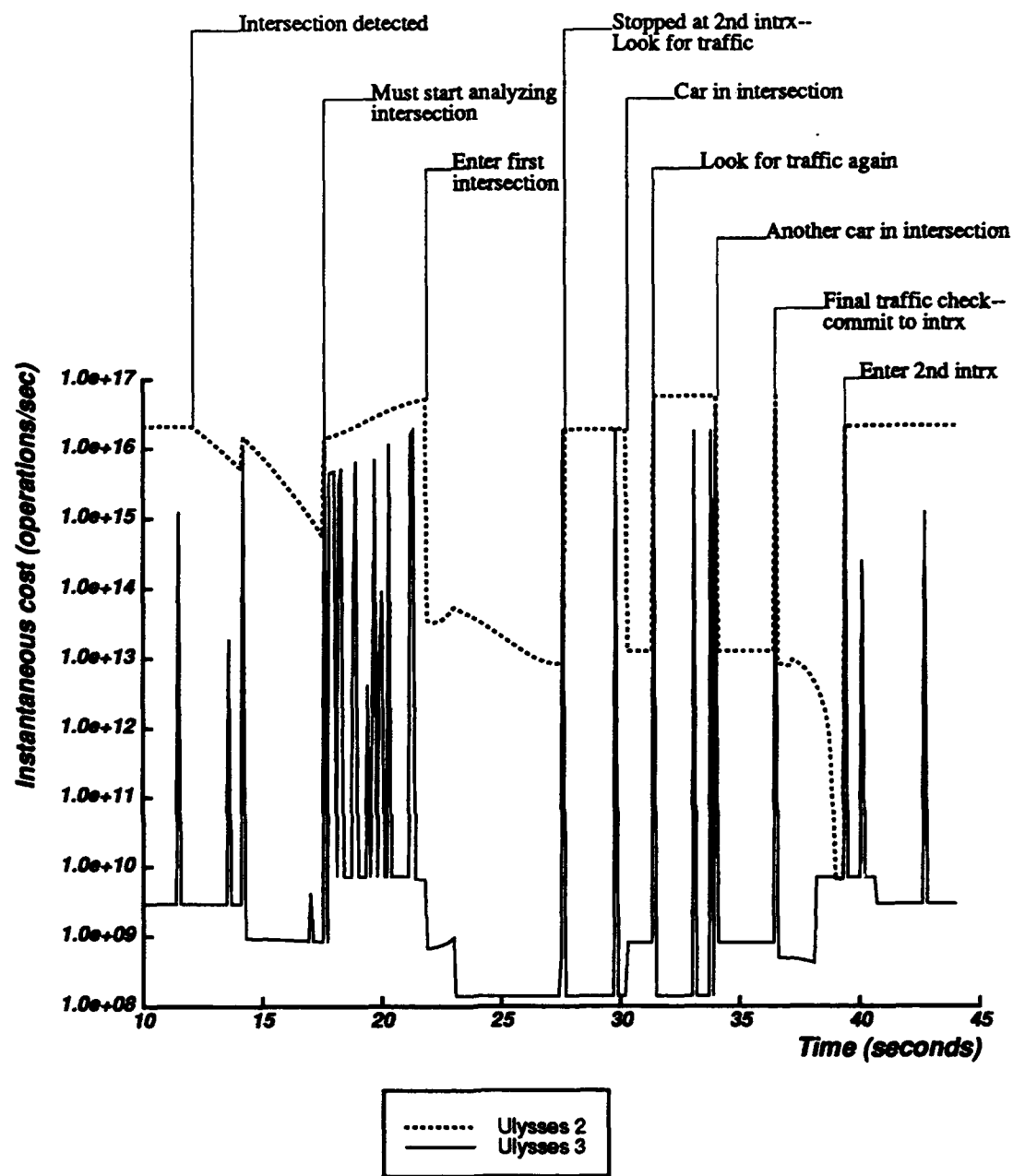


Figure 7-21: Total perceptual cost of Ulysses-3 compared to Ulysses-2 for multiple intersection scenario.

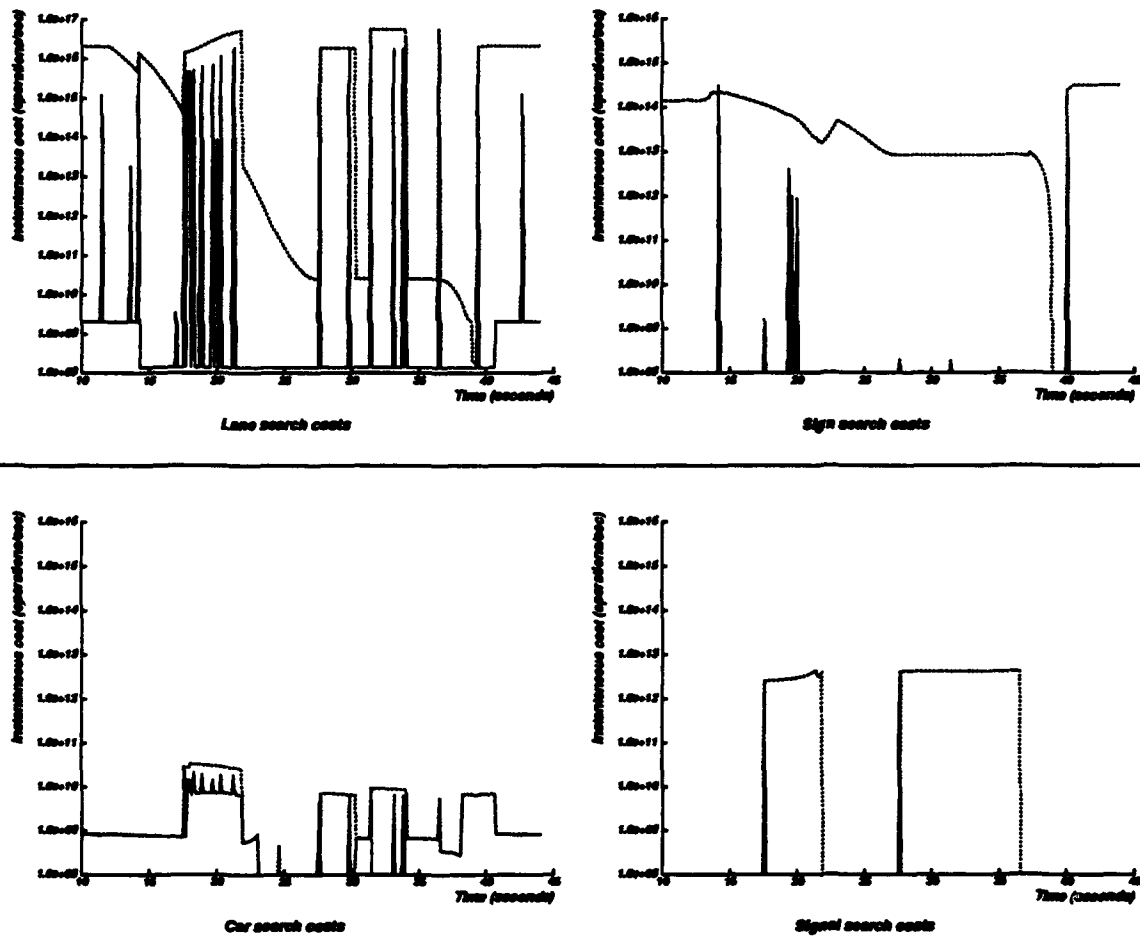


Figure 7-22: Cost of searching for various objects in Ulysses-3 (solid line), compared with Ulysses-2 (dotted line) for multiple intersection scenario.

7.4. Discussion

Net effect of selective perception. Figure 7-23 illustrates the cost of perception in the left side road scenario for naive, general perception and for all of the Ulysses implementations. On the right side of the graph, the average cost rate over the whole scenario (i.e., about 32 seconds) is given for the three implementations. The minimum perception rate, as explained in the beginning of section 7.3, is also indicated in the figure. Most of the time, Ulysses-3 requires about seven orders of magnitude less computation than Ulysses-1, and about 10 orders of magnitude less than an unguided, general perception system. When the cost of the "spikes" of Ulysses-3 is averaged over the whole scenario, Ulysses-3 is still two orders of magnitude cheaper than Ulysses-1, and over five orders of magnitude cheaper than general perception.

The approximate capabilities of some computers is also indicated in the figure. I have placed the computers a little low in the plot because the perceptual cost estimates did not really address data transfer costs, which would occupy some computer power. The "fast" computer is a hypothetical one delivering 10^9 operations per second (as suggested in the Introduction). The approximation for the Navlab with a Warp systolic array processor was taken from Clune *et al* [Clune 88]. Although the "fast" computer is almost fast enough to keep up with the requirements of Ulysses-3 most of the time, when the spikes are averaged in the cost is still too high by over five orders of magnitude. This discrepancy suggests that additional techniques should be found to reduce the cost of the spikes.

Reducing peak costs. A comparison of the instantaneous and average costs for Ulysses-3 in Figure 7-23 indicates that most of the cost is in the spikes. Furthermore, the peaks of the spikes in Ulysses-3 reach the cost levels of Ulysses-2. Thus if the robot were required to have the capacity to handle the maximum instantaneous load, it would need the same resources as the Ulysses-2 system; nothing would be gained in Ulysses-3. However, the fact that the cost in Ulysses-3 comes in spikes suggests possible solutions. For example, if the spikes could be amortized over time, the peak cost could be reduced nearly to the average cost level. This smoothing of instantaneous cost would allow the robot to carry less computation capacity and still meet load peaks.

There are probably two ways to spread out the peak costs. In Section 8.3 I describe how Ulysses-3 could implement a lazy sensing strategy that would stop sensing at a given cost threshold every decision cycle. The program would still be able to select safe actions given what it knew at the time. Naturally, we would expect the robot to slow down—i.e., performance to be decreased—as it acquires sensory data over several cycles. An alternative is to predict when costly sensing actions will be needed in the near future, and try to perform some of it in advance. This strategy would require a meta-level in the inference tree search

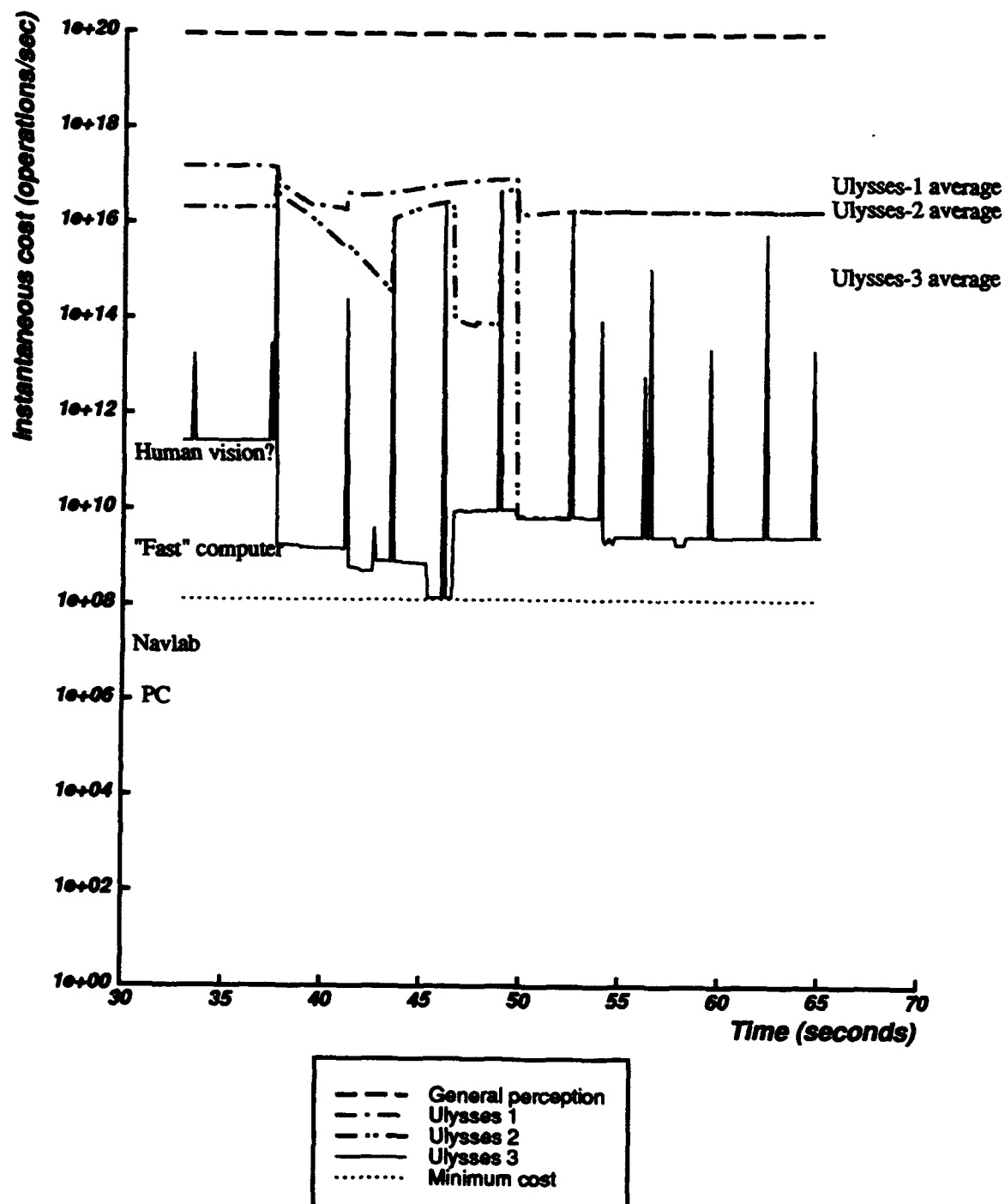


Figure 7-23:

Perceptual cost for all implementations of Ulysses in the left side road scenario. Dotted line indicates the estimated cost for finding the lane in front of the robot. This operation is not under control of Ulysses-2 or -3; thus it represents the minimum cost these implementations can achieve.

control mechanism that could hypothesize the state of the tree in future decision cycles, and predict what would constrain the robot at that time.

It is also possible to reduce the size of the cost spikes by reducing the costs of the perceptual actions used during the spike. For example, some spikes come from reacquiring objects that the robot hasn't seen for a while; it might be cheaper to introduce a *tracking* routine that could follow an object and avoid the later reacquisition. Other routines that use special assumptions could reduce the cost of specific searches.

Balancing correctness, cost, and performance. The height of the sensing peaks in Figure 7-23 could also be reduced by reducing "correctness" instead of performance. Correct driving, according to Ulysses, means always considering all constraints in the model. Ulysses is intended to generate safe actions for the robot when all constraints are applied. If, because of resource limits or *occlusions or other sensor limitations* Ulysses-3 could not determine some fact, Ulysses would allow driving performance (i.e., speed) to degrade rather than ignore a constraint. For example, if there was limited sight distance at an intersection and the robot could not see up a side road, Ulysses-3 would always assume that there could be a car on the side road prepare the robot to stop. This conservatism could get extreme; if Ulysses included the notion of children spontaneously running into the street, Ulysses-3 would force the robot to creep by all parked cars just in case a child was hiding behind one.

It is interesting to note that humans often do not reduce their performance in similar conditions. Humans clearly do not look at everything of importance in the driving environment, because they have collisions with objects that they could have avoided if they had anticipated them. There are two observations we can make about human visual search and Ulysses. First, humans probably don't have the resources needed to look for everything required by Ulysses. For example, humans have a limited field of view. The location of "Human vision" in Figure 7-23 (estimated from Moravec [Moravec 84]) also suggests that humans do much less processing than the Ulysses model requires. Second, humans make assumptions about the likelihood of various traffic situations, and selectively ignore possibilities if they seem unlikely. Thus humans can choose to ignore the possibility of children hiding behind parked cars. Human peripheral vision helps to cover some situations by flagging moving and colorful objects for attention, but it is doubtful that it can cover all of the Ulysses constraints thoroughly.

Autonomous driving models could potentially be crippled by conservative domain assumptions like those mentioned above. If a driving robot is ever to achieve human-level performance—i.e., drive at similar speeds in similar situations—it will also have to accept the risks that humans accept. Occasionally, human expectations are violated and they have accidents. This risk tradeoff could probably be incorporated into a driving program using decision theory. Sensing operations, robot speed (or lack thereof), and accidents would all

contribute to cost, and domain knowledge or learned experience would provide probability estimates for observable conditions. Thus if the cost of sensing an object was high, and in the worst case it only slowed the robot a little, then the driving program might ignore the object. If on the other hand the worst case was very slow, i.e. costly, and the best case was very unlikely to result in an accident, the program might ignore the object but assume the best case. The program could also use speed and accident cost with *a priori* probabilities to choose constraint values if the result of a sensing action was uncertain. Sensing cost, robot speed and the cost of accidents would thus affect both the robot's selection of sensing actions and its choice of driving actions in uncertain conditions.

There are several difficulties with a decision theoretic approach. First, with many constraints and many sensing options, the complexity of finding the best next action could be overwhelming. Second, probability distributions for various events and conditions could be very difficult to estimate. And finally, it is not clear what cost to assign an accident, especially one that damages property other than the robot. The driving systems described in this thesis effectively assigned infinite cost to accidents. This last question has philosophical ramifications that go beyond the technical problems.

7.5. Summary

Ulysses-3 builds on the structures and algorithms in Ulysses-2 to implement a persistent world model. The data structures are kept from decision cycle to decision cycle instead of being built from scratch. However, time stamps in the corridor show the age of each sensed data item. Ulysses-3 initializes the sense nodes of the inference tree using programs that can use the old values of the data items. If the estimate of the new state of the world is not much different than the previous state, a unique action may emerge from the root of the tree with little or no sensing. Knowledge about the dynamics of objects in the world is encapsulated in a very specific place in Ulysses-3, and the impact of these dynamics automatically falls out of the tree evaluation process.

The impact of explicitly modeling world dynamics is quite significant. The experimental results show that most of the time Ulysses-3 is spending between 5 and 7 orders of magnitude less effort than Ulysses-2 to make the same decisions. This savings is primarily the result of remembering static objects—roads, signs, and signals. Ulysses-3 also performs much better than Ulysses-2 when constraints come from somewhat predictable objects such as cross traffic. Characteristically, Ulysses-3 performs a lot of perception all at once (in "spikes") to analyze new situations, and then follows just one constraining object for as long as possible. When this object is prominent (e.g., a red light), Ulysses-3 may not have to make the big analysis again for some time. Future extensions to the perceptual routines and the inference procedure may reduce the size of the perceptual activity spikes and thus reduce the overall perceptual cost even further.

Chapter 8

Conclusions

8.1. Summary

This thesis addresses the problem of controlling perception. Complex, dynamic domains require a high perceptual bandwidth to discriminate between world states in a timely manner. For a real world problem such as driving we can demonstrate that general purpose, bottom-up scene interpretation is far too expensive. I have estimated that a computer that is fast by today's standards might still be 11 orders of magnitude too slow to do general perception. Therefore I claim that it is necessary to integrate perception and reasoning so that perceptual attention will be focused directly by the needs of the reasoning component. This thesis shows how this can be done in a principled way and illustrates the techniques in the domain of robot driving.

In order to study the driving domain, I have constructed a traffic simulator called PHAROS. This not only provides a testbed for running experiments with a simulated robot, but defines the important aspects of the street environment. The driving task was described in a computational model called Ulysses. Ulysses defines exactly how the robot should determine its actions from the observable traffic objects in the world. This model was used to implement a driving program that drives a simulated robot around in the PHAROS world. PHAROS simulates the perception and execution control of the robot as well as the environment and the actions of other drivers.

The Ulysses model was used to create a driving system in which the reasoning component actively controls perception. The system was implemented in three stages; each introduced a new technique for reducing perceptual cost. Ulysses-1 provided the basic connection between reasoning and perception that allows reasoning to request sensing actions. These actions, called perceptual routines, allow the reasoning component to request specific types of objects that are semantically important for driving decisions. At the same time they guide the perception component in its physical search for these objects. Ulysses-2 reasons about worst-case bounds on unknown conditions in the world. Ulysses-2 looks for objects in the order of their potential impact on robot actions, but stops sensing when the remaining conditions cannot affect the choice of action. The reasoning process comprises three important steps:

the initialization of sensory measurement inputs with default bounds before any perception is done; the propagation of these default bounds up an inference tree to produce a range of possible robot actions; and an exploration of this inference tree to fix just enough bounds to determine a unique action at the top. Finally, Ulysses-3 allows sensed information to be remembered, but records the age of old data. The initialization step can then use domain knowledge to adjust the bounds of sensory measurements over time. The resulting changes in certainty are again reflected in a range of robot actions at the top of the tree.

The effects of these active vision implementations are dramatic. The three versions of Ulysses were used to drive a simulated robot through five different scenarios. Ulysses-1 immediately dropped the estimated cost of perception by three to four orders of magnitude. The impact of Ulysses-2 varied more with the moment; in situations near intersections where there was often an obvious constraint on robot actions, the estimated perceptual cost dropped an *additional* three to six orders of magnitude. Ulysses-3 reduced costs even more. In most cases the estimated cost for Ulysses-3 was five to seven orders of magnitude below that for Ulysses-2, with momentary jumps every couple of seconds as the program eliminated growing uncertainty bounds. The net result is that Ulysses-3 reduced the estimated perceptual cost for driving from an intractable 10^{20} operations per second to a more reasonable 10^8 to 10^{11} operations per second.

8.2. Contributions

This thesis contributes in several ways to the design of perception for robots and to robot driving.

- **It shows how domain knowledge can be used explicitly in an active perception system to reduce perceptual costs.**
 - **Perceptual routines.** Visual routines were described in previous work; however, they dealt with lower level visual actions. Accordingly, the example domains were video game worlds. This thesis expands the concept of visual routines to a higher semantic level, and demonstrates their effectiveness in a real-world problem.
 - **Search for critical objects.** Introduces a new way to use an inference tree to determine the minimal set of objects to sense.
 - **Persistent world model.** Demonstrates a principled way to use a time-stamped world model that allows automatic determination of which facts need to be updated.
- **It shows how perception can be effectively integrated with reasoning for assessing complex situations.** Many mechanisms have been proposed for using reasoning systems to trigger perceptual actions. However, these mechanisms have focused on performing limited perceptual actions. For example, the next robot action might depend only on whether block A is on block B, where it is expected. Ulysses-3 addresses a much harder problem, in which a tremendous number of conditions in the environment could affect the robot's next action.

- It demonstrates that the traditional approach to perception for planning agents (independent, general perception) is intractable in a complex, dynamic domain like driving. Although many people have pointed out that perception and planning should be integrated, there are still vision systems being built that attempt to interpret scenes without using any information about what the agent is trying to do *right now*. This thesis provides an analysis of the environment and what it would take to interpret an arbitrary driving scene using conventional image analysis techniques.
- It introduces a new computational model of tactical driving. The Ulysses model is potentially useful not only for driving autonomous vehicles, but for studying the perceptual and decision making processes of human driving.
 - The model is computational and "complete." Previous models of human driving have not described in concrete terms how actions are computed from inputs. Other models describe details of various aspects of the driving task independently, but do not show how all aspects can be considered simultaneously.
 - The model includes perceptual tasks. Other models treat perception in general terms. Ulysses describes how to make driving decisions from observable objects.
- It illustrates the potential of simulators to model the driving task at a detailed level. Previous simulators have abstracted vehicle motions, aggregated driver decisions, and ignored the visual environment. This thesis used simulation specifically to study the perceptual load on a driver. Similar simulators could be used to study other aspects of driving and driver performance in various situations.

8.3. Future Work

This research lays the groundwork for several areas of further exploration. The Ulysses-3 system provides a structure for handling several perceptual resource problems. All of the mechanisms implemented in simulation could be improved and implemented on real systems. The driving model and PHAROS simulator could be the basis for more research in robot or human driving systems.

- **Handling perceptual constraints.** The Ulysses-3 driving system can be developed further to address other perceptual resource issues. The time-stamped database and automatic inference tree evaluation can be used to solve several resource limitation problems automatically. The following paragraphs describe capabilities that could be added to Ulysses-3 without major changes. The modified system is called "Ulysses-4."
 - **Automatic safe performance degradation with resource limitations.** Assume that the robot has the capability to handle the "steady state" perceptual costs, but not the spikes (e.g. about 10^{12} ops). We could simulate this by stopping the tree evaluation process when the cost limit was exceeded each cycle. The lower bound of the top node of the inference tree would be used as the final value. This would automatically result in a safe, pessimistic action. In later cycles, the robot would remember some things and sense different objects, and thus eventually improve all of the pessimistic default bounds. Thus, Ulysses-4 would automatically degrade the robot's performance just the right amount while it looked for everything over several cycles. This capability could be achieved with no major changes to Ulysses-3.

In some situations the robot could anticipate a cost spike and start looking ahead of time. This would require adding more knowledge to Ulysses, outside of the context of the inference tree.

- **Automatic safe performance degradation with sensing delays.** We could model delays in sensing actions due to camera field-of-view changes, processing time, resource contention, etc. The data items so affected would not be updated in the decision cycle that they were queried, although their status could change to "queried." While Ulysses-4 was waiting for the perceptual routine to return in a later decision cycle, the lower bound at the root node would be used to determine the robot's action. The inference tree would still be evaluated every cycle to age the database and make sure that the robot remains responsive to its constraints. Ulysses-4 would continually generate a safe, pessimistic action until sufficient information is available to choose a better one.

If the delayed perceptual routine does not tie up all of the perceptual resources, Ulysses-4 should probably try to sense other things while it is waiting. This additional sensing could be handled within the basic inference tree framework by adding task-specific knowledge (e.g. a Selector function, below) to the nodes. More complex resource-scheduling optimizations would require reasoning processes outside of the inference tree.

- **Improving performance.**

- **Intelligent selection of ambiguous options.** Most node functions do not have a most-critical input as Min and Max do. For example, the inference procedure has no way to select which of the (potentially) False inputs to an OR node to explore. However, we could introduce a mechanism to allow task-specific knowledge to help make a selection. A Selector function could be added to each node. When the inference procedure descends the tree to a node, this Selector function would use heuristics to pick the best input to explore. For example, at a crossroad the robot might always look for traffic to the right first. The Selector function could also be based on decision theory, using estimates of perceptual costs and the probabilities of the outcomes of perceptual actions to choose the input that is likely to be the most cost effective.
- **Learning sensing strategies.** The Selector functions describe above could be created or improved with machine learning algorithms. Whenever a node is updated, the results of the updated could be recorded in the node list (from which nodes are instantiated). The selector function could use these statistics to predict fruitful branches to explore—e.g., branches that would likely constrain the robot immediately. These statistics could also be collected for each node type *for each intersection* or other location in the world; however, this is outside the scope of the existing inference tree structure.
- **Improving inference tree mechanism.** The inference tree used in Ulysses-2 and -3 includes user-defined programs for modifying the tree during a decision cycle. The Inference Engine Module (IEM) contains task-specific node function definitions. The IEM should be expanded to include a complete set of general node functions and tree manipulation operators that have well-defined semantics. These additions would simplify the application of the inference tree mechanism to other problems.
- **Adding new routines.** Using Ulysses-3 to drive a robot will eventually establish frequently used combinations of perceptual routines. These

combinations could suggest new routines that perform some functions together. For example, since it is common to find a lane and search it only for a car, it might be efficient to have a routine that combines lane-finding and car-searching. The effort needed to identify a lane from lane lines, etc. would be reduced if the search were cut off when a car was found.

The results of Ulysses-3 also suggest new types of routines. In particular, the cost plots for Ulysses-3 suggest that tracking routines would be useful. In some cases, the dominant steady state cost comes from reacquiring dynamic objects. For example, finding the car ahead⁷ or watching a traffic signal. Special routines that tracked these objects over time would have to pay the acquisition cost only once. When these routines were applicable (e.g., when the robot is waiting at a red light), the perceptual cost would drop to the minimum value. Ulysses-3 thus concretely confirms that the active tracking systems being built by a number of researchers can be effective for real problems.

- **Trading quality for speed.** Another direction for future work is the modification of the driving model to give it higher "performance" under uncertainty in trade for degraded "correctness." In other words, when the robot can't see an object, but it can predict whether the object is there with high confidence, we could allow the robot to act on the prediction. For example humans, follow cars closely on highways under the assumption that the road exists and there are no obstacles in the lane, even though they can't always see for themselves. Human drivers also usually assume that it is safe to drive past minor side streets even when they cannot see up the street to check for traffic. This type of reasoning could be added to Ulysses by using decision theory; the program would then be able to estimate the likely outcome of ignoring various constraints.
- **Handling perceptual errors.** Another area to address is how to handle perceptual errors. The Ulysses implementations assume that perception does not miss anything important, even if this requires perception to have a very short range. False positives and short perceptual ranges can be handled automatically by the constraints. However, it is not clear what the driving model should do if there may or may not be a Stop sign ahead. The model could be modified to accept degraded "correctness," as above, and then use certainty thresholds to make decisions. This technique would require that perceptual routines have known error rates.
- **Improving PHAROS realism.**
 - Add new objects to PHAROS—e.g. bicycles, pedestrians, blinkers, driveways, other regulatory signs, and distracting objects.
 - Improve spatial reasoning—e.g. better judgement of clear paths through a congested intersection and clearance during slow lane change maneuvers.
 - Improve simulation of perception—e.g. model real sensor characteristics, limit the field of view due to occlusion, etc.
- **Extending the Ulysses driving model.**
 - Add behaviors—e.g. overtaking, creeping into an intersection, or the "Pittsburgh left turn" (a left turn across traffic at the beginning of the green light phase, rather than at the end).
 - Improve spatial reasoning—e.g. merging into a traffic flow and the spatial problems mentioned above.

⁷on a two-lane highway; on a four-lane highway, the possibility of new cars merging into the robot's lane still requires reacquisition in the current model.

- Add more heuristics about the position of lanes and their probable channelization.
- Add more heuristics for choosing lanes in congested traffic; add capability to reason abstractly about nearby traffic flow.
- Add some ability for generally recognizing the environment, e.g. to distinguish between urban and rural areas.
- Modify the style of driving based on strategic goals.

Finally, the eventual goal is to drive a real robot. This would require implementation of both the driving program and the visual routines. While implementing machine vision for a driving robot would by no means be easy, the use of the selective perception techniques described in this thesis would at least give us confidence that the problem is tractable and perhaps within the grasp of computing systems of the near future.

Appendix A

Computation Cost for the General Perception Model

In this appendix we explain our estimates of the cost of general perception. By "general" I mean a perception system that is independent of the reasoning system and must therefore naively find all (task-related) objects everywhere in the scene. Our unit of cost is an arithmetic operation on one data value or pixel. First we describe the important characteristics of the traffic objects, and then we describe the procedures used to identify them.

Traffic Object Characteristics

Minimum size and number of shape variations are the most important object characteristics for our cost calculations. For signs and signals, we also need to consider the variation in patterns within the object. Figure A-1 shows the dimensions of some traffic objects. These dimensions determine the highest angular resolution needed in an image, so we have pessimistically shown the smallest objects. For example, 20cm is the smaller of two standard sizes for traffic signal lenses.

We assume that to recognize two-dimensional objects reliably, there must be 10 pixels across the object in the image. Line objects require only 2 pixels across their width to be detected reliably. Table A-1 lists the resulting pixel resolution we require for each object. The table also gives the number of shape variations for each object. For signs, signals and markings, these are taken from lists in the MUTCD [Federal Highway Administration 78]. Car variations do not really have to be modeled in detail, because any object that is moving on the road can be considered a car; however, the naive perception system looks everywhere in the scene for vehicles, so must have detailed enough models to avoid falsely recognizing every large object.

Recognition Procedures

We assume that the general procedure for recognition will be to extract features from the image and match them to object models. Since the perception system has no guidance, it essentially has to look for all objects everywhere. We allow a few reasonable modifications, however. First, the perception system will only look for roads and road markings at negative elevation angles (i.e., below the horizon). Second, only road regions will be searched for markings. These improvements significantly reduce the pixel resolution needed in the rest of

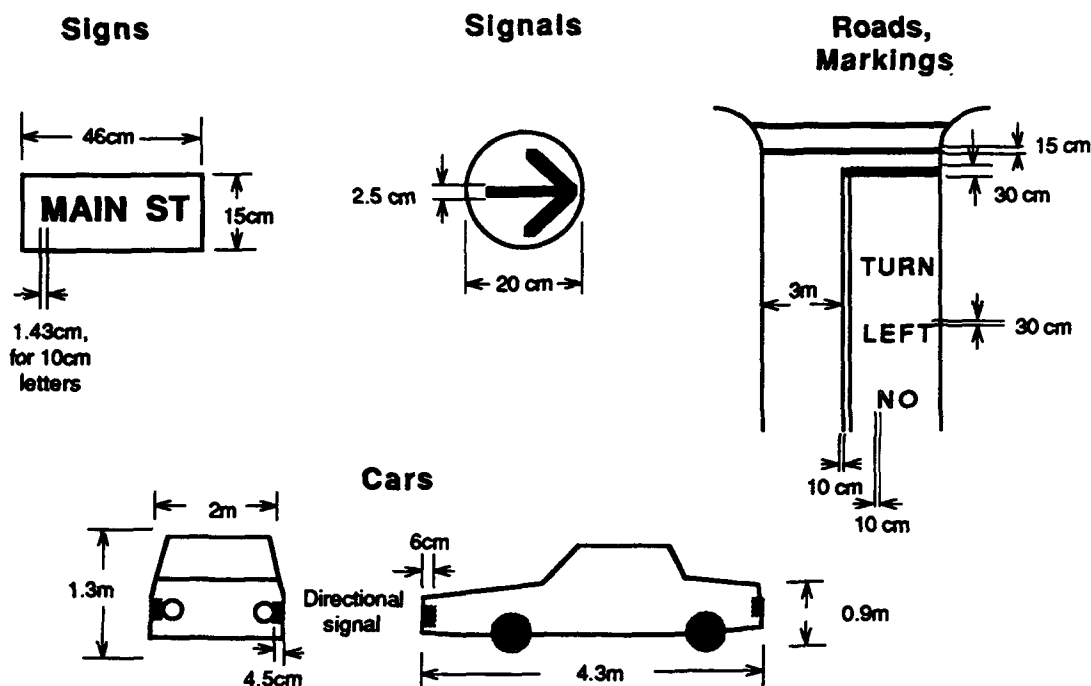


Figure A-1: Dimensions of traffic objects.

the image, since small, foreshortened road markings are the smallest objects in the scene. Third, the messages on signs and signals (e.g. letters and arrows) will only be examined after sign and signal objects have been identified. This optimization avoids having to look over the entire scene with enough resolution to read the lettering on distant signs.

The recognition procedure is thus as follows:

1. Feature extraction
2. Model matching
3. Secondary matching (detail of signs and signals)
4. Searching for road markings

The cost for recognition is the sum of the costs of these activities.

Feature Extraction. Since no one has yet built a traffic scene interpretation system, and since the environment is so complex, we do not yet know which low-level features will be necessary to recognize objects. Therefore we assume that a very wide variety of low level processing techniques must be used.

There are three general parts to feature extraction: image transformation, image segmentation, and property computation. The cost of feature extraction is the sum of the costs of these parts.

Object Type	Model Variations	Minimum resolution
Road	1	30cm
Road marking:	Lane lines- 6 (dashes, colors) Crosswalk lines Stop lines <u>Symbols and letters- 42</u> Total: 50	5cm
Signs	Octagon, triangle; Diamond- 4 (colors), <u>Rectangle- 2(colors)</u> Total: 8	1.5cm
Sign contents:	STOP, YIELD- 1 Warning- 100 Construction- 60 Regulatory- 50 Route- 40 Guide- 40 <u>Street- 36</u> Average: 50	0.7cm
Signals	Lens configurations- 10	2cm
Signal symbols:	Circle, arrows- 5	1cm
Cars	15	10cm
Turn signals:	1	3cm

Table A-1: Approximate number of model variations and resolution requirements for traffic objects. Indented object types are secondary and are considered only after the primary objects are found.

- **Image transformation.** This step includes all processes that calculate iconic features (a value at each pixel). This includes transforming color space, finding edge strength and direction, converting range data to world coordinates and local normals, transforming range and surface normals to alternate coordinates, finding gradients, and making correlations for optical flow. Each operation involves performing about the same computation on each pixel in the image. We estimate approximately 3×10^3 operations per pixel; thus

$$\text{Cost of image transformation} \approx 3 \times 10^3 P$$

where P is the number of pixels in the image.

- **Image segmentation.** After finding iconic features, they must be grouped into

semantic features such as regions, boundaries, corners, lines, etc. This process includes linking edges into lines, clustering image pixels into regions by position, color, depth, reflectance, motion, surface orientation, etc., and splitting and merging regions and lines. These steps also require some global evaluation of the quality of the segmentation using scene knowledge [Binford 82, Crisman 88, Hebert 88]. Segmentation includes three types of computations:

1. Operations on each pixel ($Cost = aP$)
2. Comparisons between pixels and regions to test membership ($Cost = bP \times \text{Number of regions}$)
3. Pairwise comparisons between regions to perform global segmentation evaluations ($Cost = c \times (\text{Number of regions})^2$)

Based on experiments we have performed with images of outdoor scenes, we believe that the number of regions is generally proportional to the number of pixels, $\text{Number of regions} = \alpha P$. This is because at higher resolutions, when there are many small pixels, more regions are visible. We can thus write an expression for total segmentation cost as

$$\text{Cost of segmentation} = aP + b\alpha P^2 + c(\alpha P)^2.$$

From experiences at CMU in perception for vehicle navigation and examination of the literature [Crisman 88, Ettinger 88, Fujimori 88, Hanson 78, Hebert 88, Kluge 88], we estimate that $\alpha \approx 10^{-2}$, $a \approx 3 \times 10^3$, $b \approx 10^2$, and $c \approx 10^2$, so the total cost for segmentation is

$$\text{Cost of segmentation} \approx 3 \times 10^3 P + P^2 + \epsilon.$$

- **Property computation.** After features are extracted, many geometrical, topological and physical properties of the features will be computed to provide further constraints for matching. These properties could include orientation, area, perimeter, moments, texture, bounding box, surface description, average color, shape description, etc. (e.g., [Fujimori 88]). Adaptive feature models—such as color classes, for example [Crisman 88]—may also be updated. The cost of these steps will be roughly proportional to the number of pixels, because many of these computations require the examination of every pixel. We estimate that

$$\text{Cost of property computation} \approx 10^3 P.$$

Adding these together, our estimate for the total cost of feature extraction is

$$\text{Cost of feature extraction} \approx 10^4 P + P^2 \tag{A.1}$$

where P is the number of pixels.

Model matching. Many reports have described the basic process of matching S scene features to M model features (for example, the survey by Besl and Jain [Besl 85]). A brute-force search through all possible pairings of features is exponentially complex— M^S in the worst case. There are many ways to reduce matching complexity using constraints, feature hierarchies, etc. (e.g. Ettinger [Ettinger 88]). However, while satisfactory computation times are often reported, the complexity of these techniques is not generally analyzed because it depends heavily on the particular situation. Grimson [Grimson 90] did rigorously analyze the complexity of recognizing certain objects using geometric constraints. He showed that the search is expected to be exponential (in M' , the visible portion of M), but he also described techniques that reduced the search cost an unspecified amount on actual problems. We have

thus found it necessary to develop a new expression to describe the actual expected cost of matching.

For our matching cost estimate we assume a sequential process in which scene features are compared to each model feature in turn (for each model in turn). Table A-2 illustrates the process. For each object model, S scene features are compared to the first model feature. Some fraction β_1 of these comparisons will result in a match after unary constraints are applied. For each of the $\beta_1 S$ matches, $(S-1)$ scene features are compared to the second model feature, for a total of $\beta_1 S(S-1)$ comparisons. Of these potential matches, a fraction β_2 will satisfy the unary and binary constraints, resulting in $\beta_1 \beta_2 S(S-1)$ valid partial (two-feature) interpretations. This process continues until $(S-(M-1))$ scene features are compared to the last model feature. Based on the enormous reductions in the search space that have been reported in the literature, we estimate that $\beta_1 \approx \beta_2 \approx 10^{-2}$, and that for later steps $\beta_i \approx 1/S$. In other words, after the second step the number of valid interpretations will not increase—for each interpretation, only one scene feature will match the current model feature m_n and satisfy all of the n -ary constraints. The total number of comparisons (for each object model)

Step	Model Feature	# Scene Features Remaining	# Comparisons	# Resulting Matches
1	m_1	S	S	$\beta_1 S$
2	m_2	$S-1$	$\beta_1 S(S-1)$	$\beta_1 \beta_2 S(S-1)$
3	m_3	$S-2$	$\beta_1 \beta_2 S(S-1)(S-2)$	$\beta_1 \beta_2 S(S-1)$
4	m_4	$S-3$	$\beta_1 \beta_2 S(S-1)(S-3)$	$\beta_1 \beta_2 S(S-1)$
...				
M	m_M	$S-(M-1)$	$\beta_1 \beta_2 S(S-1)(S-(M-1))$	$\beta_1 \beta_2 S(S-1)$

$$\text{Total (for } \beta_1 = \beta_2): \approx \beta^2 S^3 M + \beta S^2 + S$$

Table A-2:

Number of comparisons in a sequential model matching process. The number of comparisons at each step is the product of the number of valid interpretations at the previous step and the number of remaining scene features. The number of successful matches is a fraction β of these comparisons for the first two steps; for later steps, only one scene feature is successfully matched for each previous interpretation.

is obtained by adding the number in each step in the table:

$$\begin{aligned} \text{Comparisons per model} &= S + 10^{-2} S(S-1) + 10^{-4} S(S-1)((S-2) + (S-3) + \dots (S-(M-1))) \\ &\approx S + 10^{-2} S^2 + 10^{-4} S^3 M. \end{aligned}$$

In our case, the number of features is $S \approx \alpha P$ and we estimate $M \approx 20$ features per model, so the number of comparisons for a single object model is approximately $\alpha P + 10^{-2}(\alpha P)^2 + 20 \times 10^{-4}(\alpha P)^3$. Thus the cost to apply all object models is

$$\begin{aligned} \text{Cost of matching} &= N d (\alpha P + 10^{-2}(\alpha P)^2 + 20 \times 10^{-4}(\alpha P)^3) \\ &= 34 P + 3.4 \times 10^{-3} P^2 + 6.8 \times 10^{-6} P^3 \end{aligned} \quad (A.2)$$

where N is the number of objects (about 34 from Table A-1) and $d \approx 100$ is the cost of one feature comparison.

Secondary matching. As we mentioned above, after signs and signals are found the perception system must examine them in more detail to identify their contents. This involves computation similar to that described above, only for a more limited set of features (color vision only) and for only the pixels in the sign objects just found. P' for the signs found will be several orders of magnitude smaller than P for the entire scene in the expression above, so this recognition step does not add any significant cost to the overall process.

Searching for markings. The search for road markings was delayed until after road regions were found, but now the cost of finding markings must be included. In this case, unlike the case of signs and signals above, the number of pixels in the secondary search is comparable to the number in the entire scene. This difference is due to the fact that the markings are horizontal and can be greatly foreshortened at long ranges. The markings can therefore appear to be very small and require many small pixels to see (see Figure 1-5). The feature size in Table A-1 is 5cm, which requires 36 times as many pixels as are needed for the road (30cm feature); scenes from our simulated scenarios had about 3.5% of the image covered by road regions, on average. Thus we estimate that approximately the same number of pixels are used to search for markings. We assume that feature extraction would be simpler than before because the "objects" are all flat markings on the ground plane; thus

$$\text{Cost of extracting marking features} = 10^3 P + 10^{-1} P^2 \quad (A.3)$$

in contrast to Equation (A.1). The cost of matching models is calculated using Equation (A.2), except that $N = 50$ from Table A-1:

$$\text{Cost of matching markings} = 50 P + 5 \times 10^{-3} P^2 + 10^{-5} P^3. \quad (A.4)$$

The total cost of perception is then the sum of the costs in Equations (A.1) through (A.4):

$$\begin{aligned} \text{Total Cost} &= \text{Cost of feature extraction} + \\ &\quad \text{Cost of matching} + \\ &\quad \text{Cost of extracting marking features} + \\ &\quad \text{Cost of matching markings} \\ &= 1.1 \times 10^4 P + 1.1 P^2 + 1.7 \times 10^{-5} P^3. \end{aligned} \quad (A.5)$$

Pixel count. The above analysis indicates that in a scene with a fairly uniform distribution of features, the cost of perception is dependent on the number of pixels. The number of pixels is in turn dependent on the angle subtended by each pixel and the size of the field of view.

The angle subtended by a pixel depends on the size of the objects the system needs to see. Without any knowledge to constrain where objects might be, a naive perception system is forced to look for everything of potential interest in all parts of the scene. From Table A-1 we see that the smallest required feature size is 1.5cm, for finding small signs. We require that objects such as signs be recognizable even when turned away from the line of sign by 45°; thus the signs require pixels $1.5\text{cm} \times \cos(45^\circ) \approx 1.1\text{cm}$. This is for a "vertical" object; we must also consider roads, because they are horizontal and potentially foreshortened to a small size. While the 1.1cm feature covers 4.2×10^{-3} degrees at a range of 150m, a 30cm road feature covers only 1.5×10^{-3} degrees at 150m. Therefore, we assume that each pixel subtends 1.5×10^{-3} degrees.

We assume that the field of view extends 45° above and below the horizon, which allows the robot to see lines and other markings on the road within 2m of the robot, and overhead signs within several meters. The range in azimuth is the entire 360°. Thus, if the perception system uses the same resolution over the entire scene, the number of pixels will be given by

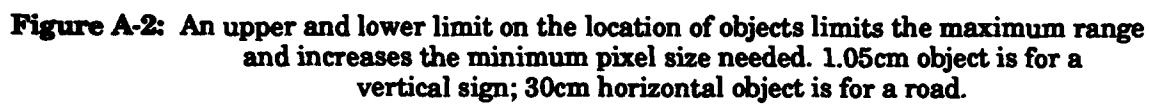
$$P = \frac{\text{Azimuth variation}}{\text{pixel angle}} \times \frac{\text{Elevation variation}}{\text{pixel angle}} = \frac{360}{1.5 \times 10^{-3}} \times \frac{90}{1.5 \times 10^{-3}} \approx 1.4 \times 10^{10}.$$

The minimum size of pixels can be increased if the range to an object is less than the maximum of 150m. We assume a flat world, and therefore can guarantee a reduced range at some elevations because of the ground plane and an assumed ceiling on traffic objects. The ceiling is determined by signs, which can be up to 7m above the road [Federal Highway Administration 78]. Figure A-2 shows how the range is limited by a floor and ceiling, and how the apparent size of an object increases when it is closer. For the lowest row of pixels in the sensor image, 45 degrees down in elevation, the pixel size corresponding to 30cm on the ground is 4°. Pixels continue to get smaller at longer ranges until at a range of 150m, 30cm on the road covers only 1.5×10^{-3} degrees. For signs, 1.1cm features vary in angle from 8.9×10^{-2} degrees to 4.2×10^{-3} degrees. Our perceptual routines take advantage of these relations to keep the resolution as low as possible. In order to keep our estimate of naive perception conservatively low, we will also apply a variable-resolution sensing technique to the pixel estimate above. We assume that sign features determine the resolution in most of the scene, except at long ranges and depressed elevations where road features appear smaller. When calculated this way, the scene requires about $P = 8.1 \times 10^7$ pixels.

It is possible to reduce the pixel count further by limiting the search for some objects to, say, the hemisphere in front of the robot. However, such gross constraints can only reduce the search cost by a small factor without using specific task knowledge.

Net cost. Using the reduced pixel count and Equation (A.5), we calculate the total cost to be

$$\text{Cost} \approx 8.9 \times 10^{11} + 7.2 \times 10^{15} + 9.0 \times 10^{18} \approx 9.0 \times 10^{18}.$$



Appendix B

The Cost of Perceptual Routines

Cost Equations

In this appendix we describe how we calculate the cost of each routine. In general we use the same assumptions about extracting features and matching models as we did for the naive model. The cost is again calculated using a polynomial in the number of pixels:

$$\text{Cost} = aP + b\alpha P^2 + Nc(\alpha P)^3,$$

where αP is the number of features in an image with P pixels, and N is the number of object models. For the naive model we used

$$\begin{aligned}\alpha &= 10^{-2}, \\ a &= 10^4, \\ b &= 10^2, \\ c &= 0.2 \\ \text{and} \\ N &= 34\end{aligned}$$

for primary feature extraction (Equation (A.1)) and model matching (Equation (A.2)). For the routines we generally use these same values. When the routines are more limited, we modify the parameters. For example, routines that search only for features on the road use $a = 10^3$ and $b = 10$ to reflect the reduced complexity of processing two-dimensional features in a plane. These are the same values used for extracting road marking features in Equation (A.3). The equation for the naive model also had terms for secondary model matching and finding road markings; these costs are also included in routines where appropriate.

The number of pixels P in the above equation depends on the geometry of the area in the world scanned by the routine. Thus the cost of each routine varies with each call. For the routines that define the corridor—track-lane, find-intersection-path, etc.—we assume that the perception system can track the road or lane in the image. That is, from a start marker where the road is identified, the routine looks in the adjacent area for the continuation of the road, and then in the area next to that, etc. Such incremental procedures are frequently used in other machine vision tasks, such as finding roads in aerial photographs [McKeown 88]. Under this assumption, the routine only searches an area slightly larger than the road or lane. Routines that search for objects such as signs or signals use a road or lane as a reference, and so can compute the entire region in the world in which to search.

Routines generally search regions that are over or next to a lane. The regions thus tend to resemble long, narrow "boxes" as in Figure B-1, with a width and length corresponding to a lane, and a height corresponding to the maximum height of the objects above the ground. The number of pixels required to cover the region depends on whether the object of interest has height (the "vertical" objects in Appendix A), or is entirely in the ground plane. In the former case, we count the number of steps necessary to sweep the entire far side of the box in one horizontal plane, as shown in Figure B-2. The number of vertical steps is approximated by dividing the region height by the object feature size. The total number of pixels is the product of horizontal and vertical steps.

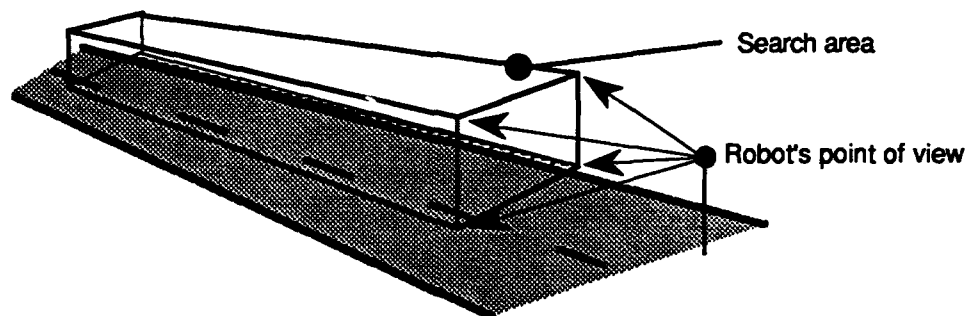


Figure B-1: When routines search within a fixed height above a lane, the resulting region is a long box.

For flat objects in the ground plane, the routines scan the the entire search area on the ground. The angle subtended by each pixel is determined by the size of foreshortened object features. For example, in tracking a lane we assume that the routine must detect both longitudinal lane lines and transverse stop and crosswalk lines. As Figure B-3 shows, either type of line may be foreshortened and limit the pixel angle, depending on the relative orientation of the road. The total number of pixels that it takes to cover the search area is the product of the number of transverse steps and the number of longitudinal steps.

For our analysis of naive perception we used the smallest resolution size from Table A-1 to determine pixel angle. Since the routines all look for different objects, they use the size appropriate for their object. Similarly, the number of model variations N depends on the particular object. The values of these parameters are taken from Table A-1. As with the naive case, we assume that signs and signals must be identified even if turned away by 45° , so the sizes given in Table A-1 are reduced by $\cos(45^\circ) \approx 0.7$.

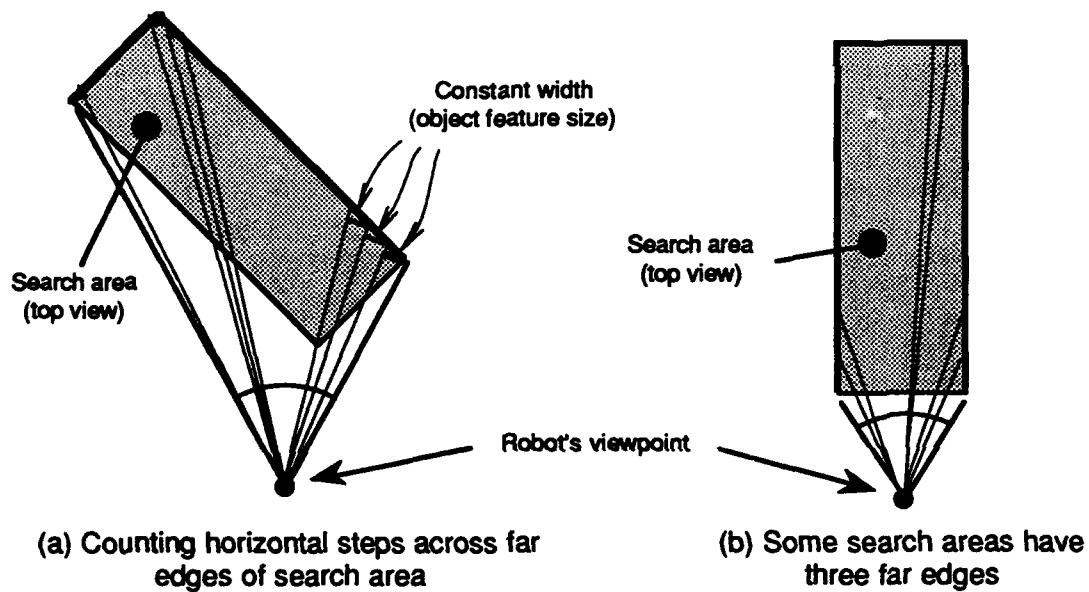


Figure B-2:

Counting horizontal steps across a search area. To see a given object feature, the routine must use small enough angular steps to see the feature even at the back of the area. For most areas (a), this involves stepping across the far two sides; for areas directly ahead of the robot (b), there are three far sides.

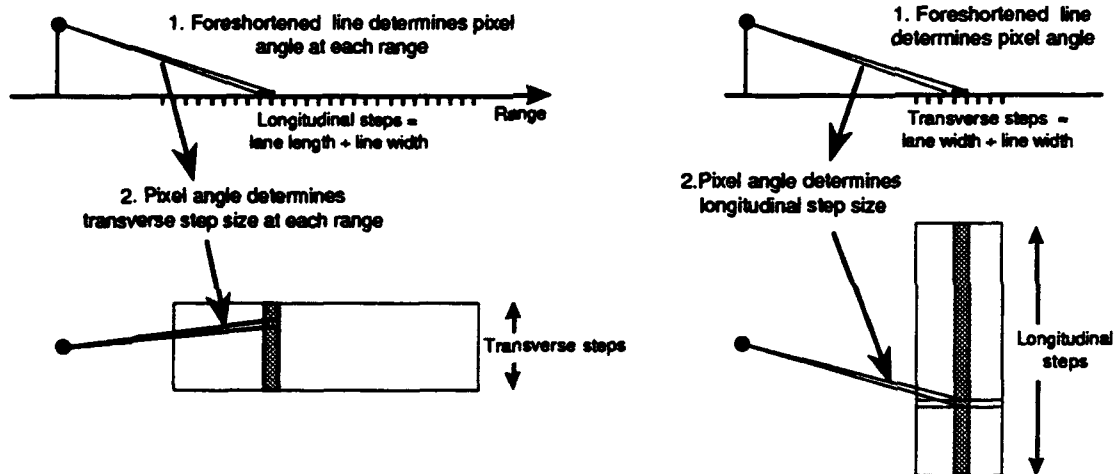


Figure B-3:

Counting the number of pixels in a search area in the ground plane. Lines in the road nearly perpendicular to the line of sight are foreshortened and determine how small the pixel angle is at each range. The total number of pixels is the product of the number of transverse and longitudinal angular steps.

Individual Routine Costs

Each of the routines uses the same cost equation with the same parameter values as the naive case except where noted.

Find current lane. This routine finds and marks the lane directly in front of the robot. It is always the first routine that Ulysses uses. From the lane line information, this routine is able to tell if the robot is between two lanes, and what angular offset there is between the robot and the lane. The robot must scan an area about 4m wide by 3m deep in front of the robot with 5cm resolution. The routine counts pixels in the ground plane, then computes the cost using $a = 10^3$, $b = 10$, and $N = 6$. The number of pixels is constant for this routine at 7580, and the cost is always 1.33×10^7 operations.

Mark adjacent lane. This routine looks for a lane adjacent to a given marker. It also searches an area about 4m by 3m for lane lines. The cost parameters are the same as above: $a = 10^3$, $b = 10$, and $N = 6$. The routine cost is thus

$$\text{Routine cost} = 10^3 P + 10^{-1} P^2 + 1.2 \times 10^{-6} P^3.$$

Track lane. This routine starts at a marker in a lane and traces the lane as far as possible. The search area is about 2m wider than the lane. The cost for track-lane is also computed using Equation .

Profile road. This routine scans transversely across the road at a marker to find all of the lanes in one direction. It reports the types of the lane lines and the relative position of the marked lane. The size of the scan area depends on the width of the road. As with the other lane-searching routines, this routine's cost is computed using Equation .

Find intersection roads. This routine finds all of the approach roads and lanes at a marked intersection. The assumed procedure is to first find intersecting roads using 30cm resolution, and then identify lanes at the intersection using 5cm resolution. The search area for the first phase is a region somewhat larger than the intersection; for the second it is the last 3m of each approach road. The parameters for the first search are the almost the same as for the lane searches above, except that $N = 1$. For the second phase the parameters are the same as for other lane searches ($a = 10^3$, $b = 10$, and $N = 6$). The cost is thus computed with

$$\text{Routine cost} = 10^3 P_1 + 10^{-1} P_1^2 + 2 \times 10^{-7} P_1^3 + \text{roads} \times [10^3 P_2 + 10^{-1} P_2^2 + 1.2 \times 10^{-6} P_2^3]$$

Find path in intersection. This routine is similar to finding intersection roads, except that it finds a departing lane in one particular direction and creates a path for the robot through the intersection. The cost is calculated using the same equation, . The number of pixels involved is smaller in the second phase of the routine because only one road is being analyzed.

Find next lane marking. This routine scans down a given lane looking for markings such as crosswalks, turn arrows, etc. The cost is just the same as tracking a lane, except that the scan area is limited to the confines of the lane, and the number of model variations $N = 44$. The cost is thus computed with

$$\text{Routine cost} = 10^3 P + 10^{-1} P^2 + 8.8 \times 10^{-6} P^3.$$

Find next car in lane. This routine scans down a given lane looking for a car. The routine doesn't have to find the lane again, so does not have to search the ground plane for lane markings. As in the naive perception case, the routine is assumed to search with low resolution first, and then search within the object for details. The low resolution search covers a region 2.5m high with 10cm pixels and uses $N = 15$. At high resolution the routine scans the area of a car (assumed to be 2.5m by 7m) with 3cm pixels and uses $N = 1$. The total cost is thus

$$\begin{aligned} \text{Routine cost} = & 10^3 P_1 + 10^{-1} P_1^2 + 3 \times 10^{-6} P_1^3 + \\ & \text{car?} \times [10^3 P_2 + 10^{-1} P_2^2 + 2 \times 10^{-7} P_2^3] \end{aligned}$$

Find crossing cars. This routine scans a marked intersection to see if there are any cars on or approaching the robot's path. The search area is limited to the intersection, and does not include approach roads. The routine searches for cars as in find-next-car-in-lane, but does not bother with the higher resolution scan of the cars found.

Find next sign. This routine searches for signs in the area to the right of the road. Because the search area is referenced to the road and not a particular lane, the routine must track the road edge. Thus the cost includes the cost of finding ground plane features, the cost of finding sign objects at low resolution, and the cost of reading the signs (secondary matching). The road search portion uses the lane tracking parameters above, except that $N = 1$ and the feature size is 30cm. The low resolution sign search covers a region about 4m wide and about 7m high along the right edge of the road. The feature size is $1.5\text{cm} \times \cos(45) \approx 1.1\text{cm}$, and $N = 8$. Reading the signs at high resolution requires seeing object features as small as $0.7\text{cm} \times \cos(45) \approx 0.5\text{cm}$, and matching $N = 50$ object models. We assume that an average sign is 91.4cm (36") on a side. This higher resolution search must examine every candidate sign that the low resolution search finds. We assume that there is a potential sign every 5m along the road. In our scenarios actual signs are spaced no closer than 30m; however, cost is calculated as if the routine examines every potential sign. The total cost is computed with

$$\begin{aligned} \text{Routine cost} = & 10^3 P_1 + 10^{-1} P_1^2 + 2 \times 10^{-7} P_1^3 + \\ & 10^3 P_2 + 10^{-1} P_2^2 + 1.6 \times 10^{-6} P_2^3 + \\ & \text{candidate signs} \times [10^3 P_2 + 10^{-1} P_2^2 + 10^{-5} P_2^3]. \end{aligned}$$

Find next overhead sign. This routine is similar to find-next-sign, except that the robot looks in the region above a given lane. Thus there is no road-finding cost. Overhead signs

are assumed to be at least 2m off the ground, and no higher than 7m. In addition, we assume that the low resolution search finds potential signs every 20m instead of every 5m. We expect there to be fewer objects extending over a lane than there are erected beside the road. The equation for computing the routine cost is

$$\text{Routine cost} = 10^3 P_2 + 10^{-1} P_2^2 + 1.6 \times 10^{-6} P_2^3 + \\ \text{candidate signs} \times [10^3 P_2 + 10^{-1} P_2^2 + 10^{-5} P_2^3].$$

Find back-facing signs. This routine looks for STOP and YIELD signs along the left side of the road, facing opposing traffic. The robot only looks about 25m up the road before giving up. The cost of the routine includes a road-edge tracking cost, as with find-next-sign, and a low resolution sign search cost. A higher resolution scan to read signs is not necessary since they are facing away from the robot. For the road, the feature size is again 30cm and $N = 1$. When searching for the signs, the robot only needs to look for two shapes—a triangle or an octagon—so $N = 2$. STOP and YIELD signs are at least 76cm \times 76cm, so the feature size is $7.6\text{cm} \times \cos(45^\circ) = 5.4\text{cm}$. The equation for computing the cost is thus

$$\text{Routine cost} = 10^3 P_1 + 10^{-1} P_1^2 + 2 \times 10^{-7} P_1^3 + \\ 10^3 P_2 + 10^{-1} P_2^2 + 4 \times 10^{-7} P_2^3$$

Find signal. This routine scans a marked intersection to find signal heads facing the robot. The search region covers the whole area of the intersection because signals can be on the near or far side, and on the left or right. The routine first finds the intersection area using 30cm features and parameter values $a = 10^3$, $b = 10$, and $N = 1$. Next there is a low-resolution search for signal heads using $2\text{cm} \times \cos(45^\circ) = 1.4\text{cm}$ features, and $N = 10$. Finally, the lenses found in the low-resolution search are examined at higher resolution to detect arrows. This process uses 0.7cm features and $N = 5$. We assume that there are three lenses on each signal head at the intersection, and each is examined to determine what symbols it displays. Each lens is assumed to be 30cm in diameter. The total cost of the routine is given by

$$\text{Routine cost} = 10^3 P_1 + 10^{-1} P_1^2 + 2 \times 10^{-7} P_1^3 + \\ 10^3 P_2 + 10^{-1} P_2^2 + 2 \times 10^{-6} P_2^3 + \\ \text{lenses} \times [10^3 P_2 + 10^{-1} P_2^2 + 10^{-6} P_2^3].$$

Marker distance. This routine calculates the (distance in the world, not an image) between two markers. It is used primarily to estimate the distance of another car to the intersection it is approaching. We assume that the perception system records world coordinates of each marker it creates, so this "routine" does not involve any perception at all. The cost is therefore zero.

Appendix C

Routine Costs in the Left Side Road Scenario

This appendix shows in more detail the cost of performing various perceptual routines in a specific situation. The situation we illustrate is a moment in the left side-road scenario, shown again in Figure C-1. This is a point in time between notes (4) and (5) in Figure 5-10. The use of routines here was already described by Figures 5-3 through 5-8 in Chapter 5; the planner uses (only) a model of the driving corridor to select perceptual targets. We repeat those figures here and list the estimated cost (in operations) of each perceptual action. Note that this cost is for a 100 millisecond cycle, and so is a factor of 10 smaller than the cost per second plotted in Chapter 5.

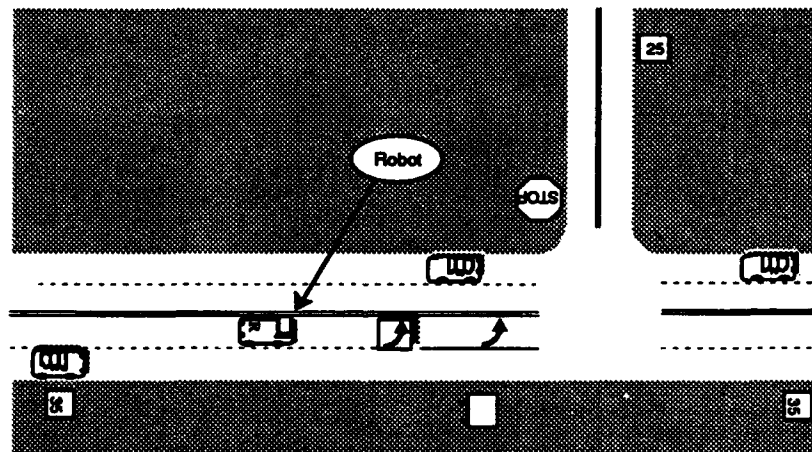
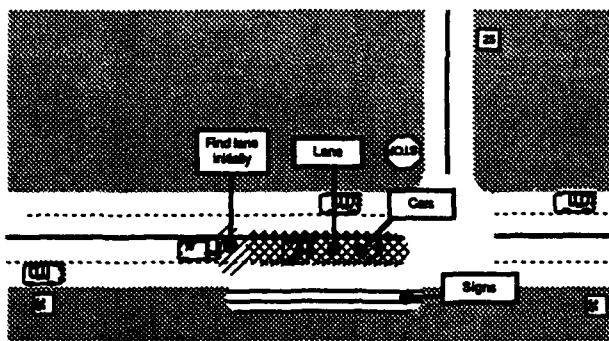


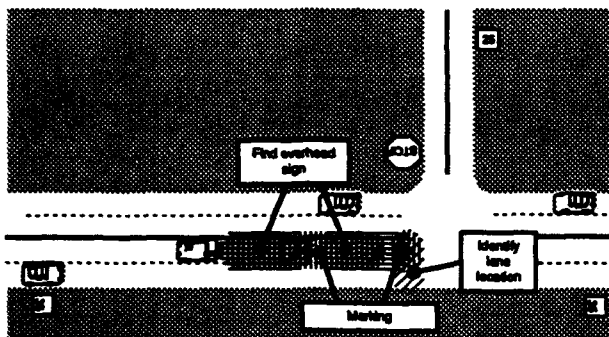
Figure C-1: The moment of the left side-road scenario analyzed in this appendix.

Most actions correspond to single perceptual routine calls, except where noted. Multiple routine calls are made when the first scan terminates on an object, requiring another call to scan the rest of the corridor.

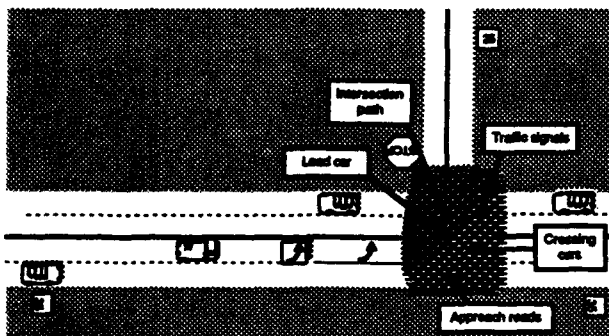
Several costly actions have been highlighted in bold face. These actions dominate the perceptual cost at this moment in the scenario. The actions all involve tracking lanes—i.e., looking for lanes bounded by narrow lines. The search for lanes is expensive in several cases because the lanes extend all of the way to the range limit of the sensors (150m), where very high resolution is needed to resolve the lines.



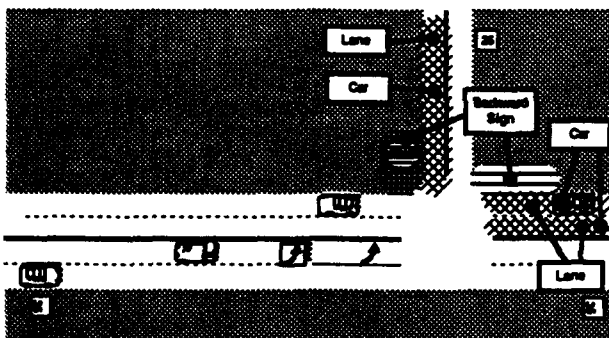
Find lane initially: 1.3×10^7
 Track lane: 5.0×10^{12}
 Look for car: 6.6×10^7
 Look for signs (2 scans): 1.1×10^{13}



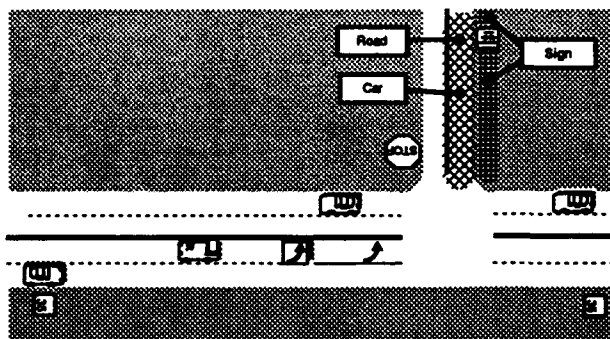
Find lane location: 5.0×10^{12}
 Find overhead signs (2 scans): 1.1×10^{12}
 Find markings (2 scans): 4.6×10^{12}



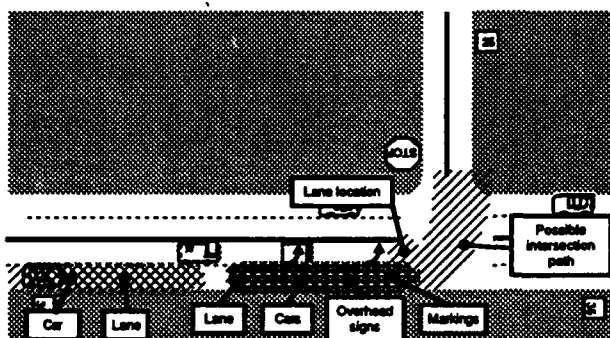
Find path in intersection: 2.2×10^8
 Look for crossing cars: 1.9×10^8
 Look for signals: 2.8×10^{12}
 Find approach roads: 1.7×10^{12}
 Look for lead car: 6.4×10^8



Find back-facing signs
 (2 approaches): 3.7×10^8
 Track lanes upstream
 (3 lanes): 1.8×10^{15}
 Look for approaching cars
 (in 3 lanes): 1.3×10^9



Track lane: 4.7×10^{14}
 Look for cars: 6.6×10^8
 Find speed limit signs (2 scans): 9.5×10^{13}



Find and track adjacent lane: 4.8×10^{12}
 Look for cars ahead: 7.4×10^7
 Find lane location at intersection: 5.0×10^{12}
 Look for signs above lane: 1.9×10^{12}
 Look for markings in lane: 1.0×10^{13}
 Look for intersection path: 9.3×10^{10}
 Track lane upstream: 2.0×10^{15}
 Find car upstream: 6.6×10^8

Appendix D

Bounds Propagation Algorithms

This appendix describes how bounded values ("intervals") are propagated through the Ulysses-2 and -3 inference trees. First some of the restrictions on functions and symbolic values are presented, and then propagation formulas for all Ulysses functions are listed.

Monotonicity

All functions in the inference trees must be monotonic. That is, for a function $f(x)$, either

$$f(x_1) \leq f(x_2) \text{ when } x_1 < x_2,$$

or

$$f(x_1) \geq f(x_2) \text{ when } x_2 < x_1.$$

The functions are required to be monotonic so that the extreme values of the functions can be found from just the upper and lower bounds of their inputs. The formulas for doing this are described below.

Bounds on Symbol Values

The meaning of bounded intervals is clear for numbers, but intervals are not always meaningful for symbols. In Ulysses-2 and -3, however, when an input to a function is symbolic, the values that the input can take are restricted to make intervals meaningful.

There are several types of restrictions:

- **Booleans.** Booleans inputs can have only two values, so if the input value is not known the upper bound must be True and the lower bound False. (The upper bound is not allowed to be False when the lower bound is True.)
- **Ordered symbols.** Sometimes it may be possible to assign a total ordering to a set of symbols. Since the symbols can then be mapped onto a subset of integers, we can use bounded intervals of symbols the same way we would use intervals of integers. The Make-tcd-class function below uses ordered symbols.

Note that the ordering of the symbols may not come from an inherent property of the symbols, but from domain semantics. For example, the *signal* input to Make-tcd-class is ordered by decreasing likelihood of stopping the robot (Red, Amber, Green), which in turn implies increasing allowed acceleration.

- **"Monotonic sets."** When an input represents a set, we can place restrictions on the values taken by the set-input so that we can order them. For example, let the elements of a set be taken from domain D . Suppose we restrict the values X_i taken by a set-input as follows:

$$\begin{aligned}
X_0 &= \emptyset \\
X_1 &= X_0 \cup \text{some elements of } D \\
X_2 &= X_1 \cup \text{some other elements of } D \\
&\dots \\
X_n &= D
\end{aligned}$$

We can then define an ordering on the values as follows:

$$X_i \leq X_j \text{ iff } X_i \subseteq X_j.$$

I call inputs with such restricted values *monotonic set* inputs. Note that the empty set (represented by *nil* in Ulysses) is the lowest possible lower bound, and the universal set (represented by *true*) is the highest upper bound.

For example, the Member-p function described below assumes that its set input is a monotonic set input. Let Y be the monotonic set input, with L_Y the lower bound of Y and U_Y the upper bound. Let x be the symbol being tested for membership. We know that $L_Y \subseteq U_Y$. If $x \in L_Y$, then $x \in Y$, and if $x \notin U_Y$, then $x \notin Y$. Thus L_Y and U_Y are useful lower and upper bounds for Y .

Function Descriptions

Three pieces of information are given for each function:

- The function definition—i.e., how the function value would be computed if the inputs were single values instead of intervals.
- Special restrictions or assumptions for the function. For example, domain restrictions.
- How the upper and lower bounds are computed.

The functions are defined in terms of input variables such as p, x_i , etc. The upper and lower bounds of the variables are indicated by the symbols U and L with subscripts to identify the input, e.g. $U_p, L_p, U_{x_i}, L_{x_i}$.

Not

- $f(x) = \neg x$
- Input and output are boolean values.
- $Upper\ bound = \neg L_x$
 $Lower\ bound = \neg U_x$

Or

- $f(x_1, x_2, \dots) = x_1 \vee x_2 \vee \dots$
- Inputs and output are boolean values.
- $Upper\ bound = U_{x_1} \vee U_{x_2} \vee \dots$
 $Lower\ bound = L_{x_1} \vee L_{x_2} \vee \dots$

And

- $f(x_1, x_2, \dots) = x_1 \wedge x_2 \wedge \dots$
- Inputs and outputs are boolean values.

- $Upper\ bound = U_{x_1} \wedge U_{x_2} \wedge \dots$
 $Lower\ bound = L_{x_1} \wedge L_{x_2} \wedge \dots$

>

- $f(x, y) = (x > y)$
- Inputs are real values; output is boolean.
- $Upper\ bound = U_x > L_y$
 $Lower\ bound = L_x > U_y$

=

- $f(x, y) = (x = y)$
- Inputs are real values; output is boolean.
- **If** $(L_x > U_y) \vee (L_y > U_x)$
 $Upper\ bound = False$
 $Lower\ bound = False$
- **Else if** $U_x = L_x = U_y = L_y$
 $Upper\ bound = True$
 $Lower\ bound = True$
- **Else**
 $Upper\ bound = True$
 $Lower\ bound = False$

If-num

- $f(p, x, y) = \begin{cases} x & p = True \\ y & p = False \end{cases}$
- Conditional numerical switch (referred to as "if-then-else" in the text).
Input p is boolean, but x and y are real numbers.
- **If** $U_p = L_p$
 $Upper\ bound = \begin{cases} U_x & p = True \\ U_y & p = False \end{cases}$
 $Lower\ bound = \begin{cases} L_x & p = True \\ L_y & p = False \end{cases}$
- **Else**
 $Upper\ bound = Max\ of\ \{U_x, U_y\}$
 $Lower\ bound = Min\ of\ \{L_x, L_y\}$

Max (and Min)

- $f(x_1, x_2, \dots) = \text{Maximum of } \{x_1, x_2, \dots\}$
- Inputs are real numbers. Note: the Min function is the same, except the minimum of inputs is taken.
- *Upper bound* = Maximum of $\{U_{x_1}, U_{x_2}, \dots\}$
Lower bound = Maximum of $\{L_{x_1}, L_{x_2}, \dots\}$

+

- $f(x_1, x_2, \dots) = x_1 + x_2 + \dots$
- Inputs and output are real numbers.
- *Upper bound* = $U_{x_1} + U_{x_2} + \dots$
Lower bound = $L_{x_1} + L_{x_2} + \dots$

-

- $f(x, y) = x - y$
- Function of two real numbers returns a real number.
- *Upper bound* = $U_x - L_y$
Lower bound = $L_x - U_y$

×

- $f(x, y) = x \times y$
- Function of two real numbers returns a real number.
- *Upper bound* = Max of $\{U_x \times U_y, U_x \times L_y, L_x \times U_y, L_x \times L_y\}$
Lower bound = Min of $\{ \text{the same} \}$

fn-1

- $f(x) = \text{defined by user}$
- Arbitrary monotonic function of one argument. Function may be monotonically increasing or decreasing. Input can be of any type; output is a real number.
- *Upper bound* = Max of $\{f(U_x), f(L_x)\}$
Lower bound = Min of $\{f(U_x), f(L_x)\}$

fn-2

- $f(x, y) = \text{defined by user}$
- Arbitrary monotonic function of two arguments. Inputs can be of any type; output is a real number.
- *Upper bound* = Max of $\{f(U_x, U_y), f(U_x, L_y), f(L_x, U_y), f(L_x, L_y)\}$
Lower bound = Min of $\{f(U_x, U_y), f(U_x, L_y), f(L_x, U_y), f(L_x, L_y)\}$

Accel-eqn

$$f(v_r, v_c, g) = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \text{ where}$$

$$A = \Delta T^2,$$

$$B = \Delta T (2v_r - d \Delta T),$$

$$C = (v_r^2 - v_c^2) + 2d(g - v_r \Delta T)$$

v_r is the robot's speed,

v_c is the constraint speed,

g is the gap to the constraint,

ΔT is the decision cycle time, and

d is the nominal deceleration rate of the robot.

- This is the function used to compute an acceleration constraint given the robot's speed, the constraint speed, and the distance to the constraint (see Figure 3-2). All input values are assumed to be positive.

$$\text{Upper bound} = f(L_{v_r}, U_{v_c}, U_g)$$

$$\text{Lower bound} = f(U_{v_r}, L_{v_c}, L_g)$$

Car-accel-eqn. This function is the same as accel-eqn above, except that

$$C = (v_r^2 - \frac{d}{d'} v_c^2) + 2d(g - v_r \Delta T)$$

where d' is a high rate of deceleration. This function computes the constraint generated by a car in front of the robot; it is assumed that this car could brake at rate d' .

Eq1

$$f(x, y) = (x = y)$$

- Symbolic equal predicate. Both inputs are symbols; the output is a boolean. The Lisp eq1 function is used to make the test.

$$\text{If } (L_x = U_x) \wedge (L_y = U_y) \\ \text{Upper bound} = (x = y) \\ \text{Lower bound} = (x = y)$$

Else

$$\text{Upper bound} = \text{true}$$

$$\text{Lower bound} = \text{false}$$

If-sym

- $f(p, x, y) = \begin{cases} x & p = \text{True} \\ y & p = \text{False} \end{cases}$
- Conditional symbolic switch. Input p is boolean. When p is uncertain, *if-sym* must find the extremes of the inputs x and y ; however, the *if-sym* function contains no knowledge of how the symbol values are ordered, so it can only pick one bound arbitrarily. *If-sym* does assume, however, that the value *nil* is an extreme low value. (Note that we could define *special* versions of *if-sym* that compared x and y to find the global upper and lower bounds. Also, if x and y were monotonic sets, $\text{Max}(x, y)$ is $U_x \cup U_y$, and $\text{Min}(x, y)$ is $L_x \cap L_y$)
- **If** $U_p = L_p$
$$\text{Upper bound} = \begin{cases} U_x & p = \text{True} \\ U_y & p = \text{False} \end{cases}$$

$$\text{Lower bound} = \begin{cases} L_x & p = \text{True} \\ L_y & p = \text{False} \end{cases}$$

Else
$$\text{Upper bound} = \begin{cases} U_x & U_x \neq \text{nil} \\ U_y & U_x = \text{nil} \end{cases}$$

$$\text{Lower bound} = \begin{cases} L_x & L_x \neq \text{nil} \\ L_y & L_x = \text{nil} \end{cases}$$

Fn-1-sym

- $f(x) = \langle \text{defined by user} \rangle$
- This is a function of one argument. The input can be anything, including a symbol; the output may be any type. The function is assumed to be monotonically increasing.
- $\text{Upper bound} = f(U_x)$
 $\text{Lower bound} = f(L_x)$

Fn-x-sym

- $f(x_1, x_2, \dots) = \langle \text{defined by user} \rangle$
- This is an extension of the above function to any number of arguments. The inputs and output can be of any type (normally, symbols). The function is assumed to be monotonically increasing in all inputs.
- $\text{Upper bound} = f(U_{x_1}, U_{x_2}, \dots)$
 $\text{Lower bound} = f(L_{x_1}, L_{x_2}, \dots)$

Intersection

- $f(x_1, x_2, \dots) = x_1 \cap x_2 \cap \dots$
- Set intersection. The inputs and output are monotonic sets.
- $Upper\ bound = U_{x_1} \cap U_{x_2} \cap \dots$
 $Lower\ bound = L_{x_1} \cap L_{x_2} \cap \dots$

Member-p

- $f(x, y) = (x \in y)$
- Membership predicate. x is a symbol, and y is a monotonic set.
- $Upper\ bound = (U_x \in U_y) \vee (L_x \in U_y)$
 $Lower\ bound = (U_x \in L_y) \vee (L_x \in L_y)$

Fn-1-set

- $f(x) = \langle \text{defined by user} \rangle$
- A user-defined function of one argument. There are no restrictions here on the type of the argument; the user must make sure that the input type matches his function's input requirements. The output of the function is a set. The function must be monotonic in that either
$$f(x_1) \subseteq f(x_2) \text{ if } x_1 < x_2$$
(monotonically increasing), or
$$f(x_1) \subseteq f(x_2) \text{ if } x_2 < x_1$$
(monotonically decreasing). For symbolic inputs, the function and the inputs must be restricted so that
$$f(L_x) \subseteq f(U_x).$$
- If $f(L_x) \subseteq f(U_x)$
$$Upper\ bound = f(U_x)$$
$$Lower\ bound = f(L_x)$$

Else
$$Upper\ bound = f(L_x)$$
$$Lower\ bound = f(U_x)$$

Fn-2-set

• $f(x, y) = \langle \text{defined by user} \rangle$

- An extension of Fn-1-set to two arguments. A straightforward extension would normally require that the function be monotonic in both variables at once; for Ulysses, however, the extension was allowed to be less restrictive. The function is only required to be monotonic in one variable at a time. That is, the function must be monotonic in one variable while the other variable is fixed—e.g.,

$$f(x_1, Y) \subseteq f(x_2, Y) \text{ if } x_1 < x_2.$$

If neither input is fixed (i.e., the upper bound equals the lower bound), then Fn-2-set always sets its upper bound to *true* and its lower bound to the empty set.

```
• If  $L_x = U_x$ 
  If  $f(x, L_y) \subseteq f(x, U_y)$ 
    Upper bound =  $f(x, U_y)$ 
    Lower bound =  $f(x, L_y)$ 
  Else
    Upper bound =  $f(x, L_y)$ 
    Lower bound =  $f(x, U_y)$ 
Else if  $L_y = U_y$ 
  If  $f(L_x, y) \subseteq f(U_x, y)$ 
    Upper bound =  $f(U_x, y)$ 
    Lower bound =  $f(L_x, y)$ 
  Else
    Upper bound =  $f(L_x, y)$ 
    Lower bound =  $f(U_x, y)$ 
Else
  Upper bound = true
  Lower bound = {nil}
```

Make-tcd-class

• $f(r, m, s, l) = \langle \text{special function} \rangle$

- Make-tcd-class is a special function that computes a TCD class from four inputs: the range to the intersection; the robot's intended maneuver at that intersection; the traffic control sign for the robot at that intersection; and the traffic signal facing the robot at that intersection. The function is described by the left side of Table 3-1 ("minor road" tests are added elsewhere in the inference tree).

The inputs to this function are symbols. It is possible to have upper and lower bounds in this case because the symbols can be ordered. The *range* is a number, so can trivially be ordered. The *maneuver* is only important if it is a right turn, so there are only two equivalence classes for the maneuver; essentially, the bounds can be *Right* or $\neg \text{Right}$. The *sign* has three ordered values: {Stop, Yield, Other}. The *signal* input has four ordered values: {Red, Amber, Green, None}. Because the input symbols can be ordered, the function can compute upper and lower bounds on the TCD class. For example, a high *range*, a $\neg \text{Right}$ turn *maneuver*, a Stop *sign*, and a red *signal* would result in the lowest priority TCD class in Table 3-1. Values at the other extremes would result in the highest priority class.

Lane-select

- $f(x_1, x_2, \dots) = m(\text{Max of } \{x_1, x_2, \dots\})$
 where $m(x)$ gives the lane choice (symbol) that corresponds to the lane preference rule with priority x . A larger x indicates higher priority.
- This is the top level function for lane selection. Lanes are selected by considering rules that express preferences for different lanes (either the lane to the left, the lane to the right, or the current lane). Each rule has a priority value. The applicable preference rule with the highest priority determines to which lane the robot will move.

The inputs to Lane-select are priority values (numbers). Each priority value corresponds to a preference rule, which in turn corresponds to a lane choice. The special priority value "-1" does not correspond to any lane; this is the priority used for preference rules that are not applicable in the current situation. Thus the bounds on the inputs are in general $[-1, x]$, where x is the normal priority value of the preference rule.

Lane-select performs the exact same function as Max (described above), except that it maps the output value into a lane choice. Furthermore, if at any point all the applicable preference rules correspond to the same lane choice, Lane-select produces that choice.

- $\text{Upper bound} = m(\text{Max of } \{U_{x_1}, U_{x_2}, \dots\})$

$$\text{Lower bound} = \begin{cases} m(\text{Max of } \{L_{x_1}, L_{x_2}, \dots\}) & \text{Otherwise} \\ m(U_x) & \text{One action available} \end{cases}$$

"One action available" is true when

for all $i: U_{x_i} \neq -1$,

$$m(U_{x_i}) = A$$

where A is some action.

Appendix E

Input Selection Algorithms

This appendix contains the algorithms for choosing node inputs for exploration during the Ulysses-2 and -3 search procedure. This search procedure is used to find the most constraining sense input to the tree. The procedure starts at the root of the tree and descends recursively, always trying to choose the branch that could uniquely determine the value of the tree. For example, at a Max node the search algorithm would explore the input with the highest upper bound. It is not always possible to select a "best" branch to explore; in these cases, the first input is chosen arbitrarily. The selection algorithm for each function (except single-argument functions) used in the inference trees is given below.

Cutoffs

In the spirit of branch-and-bound algorithms, some functions generate cutoff values that can be used to stop further tree search. There are three types of cutoffs: *max*, *min*, and *equal*. A *max* cutoff is a value above which the subtree value cannot rise before search is stopped. A Min node generates a *max* cutoff. For example, suppose that one input to the Min node was bounded by 4 and 5 (which we can write as the interval [4, 5]), while the second input was [1, 8]. The second input should be explored first because it contains the minimum value. However, exploration should proceed with a *max* cutoff value of 5. That is, if the lower bound of the second input ever rises above 5, exploration of that subtree should stop. In that case the node value must come from the first input and the second is no longer important. Similarly, a *min* cutoff is a value below which the subtree value cannot sink before search is stopped. And *equal* cutoff is a value that must be included in the bounded value of the subtree. The list below also includes the algorithms for generating cutoffs, where appropriate.

The cutoff value generated at a node is sometimes a function of the cutoff value passed down into that node from its parent. For example, the If-num function below just generates the same cutoff that was passed down to the node. In the list below, the value of the cutoff passed down from the parent is given by *C*, and the type of the cutoff is given by *T*.

Selection Algorithms

In the list below, U_x is the upper bound of the input x , and L_x is the corresponding lower bound. The information for each function includes the function definition, the algorithm for choosing an input and generating a cutoff, and explanatory notes if necessary.

p

Or, And

- $f(x_1, x_2, \dots) = x_1 \vee x_2 \vee \dots$ (similarly for And)
- Choose the first input x with $U_x \neq L_x$.
- No cutoff generated.

>

- $f(x, y) = (x > y)$
- If $U_x \neq L_x$
 - Choose x ;
 - Generate *min* cutoff = L_y
- Else
 - Choose y ;
 - Generate *max* cutoff = U_x

=

- $f(x, y) = (x = y)$
- If $U_x \neq L_x$
 - If $U_y = L_y$
 - Choose x ;
 - Generate *equal* cutoff = y
 - Else
 - Choose x ;
 - Generate *max* cutoff = U_y
- Else
 - Choose y ;
 - Generate *equal* cutoff = x
- If both x and y have uncertain values, the first input (x) is chosen arbitrarily. In this case the search could actually make use of two cutoffs: a max cutoff with the value U_y , and a min cutoff with the value L_y . However, the algorithm as implemented does not accomodate two cutoffs, so the max cutoff is used alone.

If-num

$$\bullet f(p, x, y) = \begin{cases} x & p = \text{True} \\ y & p = \text{False} \end{cases}$$

$$\bullet \text{If } U_p = L_p$$

 If p

 Choose x ;

 Pass down T cutoff = C

 Else

 Choose y ;

 Pass down T cutoff = C

 Else

 Choose p ;

- If p is uncertain, the p input may or may not actually be the best branch to explore. Its selection is somewhat arbitrary. For example, if $x = [1, 5]$, $y = [4, 6]$ and there is a max cutoff $C = 2$, then it might be easier to explore x to raise its lower bound above the cutoff. However, there is no way to know this ahead of time without more information about the subtrees.

Max

$$\bullet f(x_1, x_2, \dots) = \text{Maximum of } \{x_1, x_2, \dots\}$$

$$\bullet \text{Choose input } x_n: U_n = \text{Max of } \{U_{x_1}, U_{x_2}, \dots\}$$

$$\text{Generate } \min \text{ cutoff} = \text{Max of } L_{x_1}, L_{x_2}, \dots$$

- The exact algorithm is used for function **Lane-select**.

Min

$$\bullet f(x_1, x_2, \dots) = \text{Minimum of } \{x_1, x_2, \dots\}$$

$$\bullet \text{Choose input } x_n: U_n = \text{Min of } \{U_{x_1}, U_{x_2}, \dots\}$$

$$\text{Generate } \max \text{ cutoff} = \text{Min of } U_{x_1}, U_{x_2}, \dots$$

+

$$\bullet f(x_1, x_2, \dots) = x_1 + x_2 + \dots$$

$$\bullet \text{Choose the first input } x_n: U_{x_n} \neq L_{x_n}$$

 If $T = \max$

$$\text{Generate } \max \text{ cutoff} = C - \sum L_{x_i}, i \in \text{inputs}, i \neq n$$

 Else if $T = \text{equal}$

 If there was only one input x with $U_x \neq L_x$,

$$\text{Generate } \text{equal} \text{ cutoff} = C - \sum L_{x_i}, i \in \text{inputs}, i \neq n$$

 Else

$$\text{Generate } \max \text{ cutoff} = C - \sum L_{x_i}, i \in \text{inputs}, i \neq n$$

 Else if $T = \min$

$$\text{Generate } \min \text{ cutoff} = C - \sum U_{x_i}, i \in \text{inputs}, i \neq n$$

- If no cutoff is passed to this node, none is generated. If T is *equal*, both a *max* and a *min* cutoff could be generated and passed down the tree. However, the search algorithm as implemented does not record two cutoff values, so a *max* cutoff is used arbitrarily.

- $f(x,y) = x - y$
- If $T = \text{max}$
 - If $U_x \neq L_x$
 - Choose x
 - Generate *max* cutoff = $C + U_y$
 - Else
 - Choose y
 - Generate *min* cutoff = $L_x - C$
- Else if $T = \text{equal}$
 - If $U_x \neq L_x$
 - Choose x
 - Generate *equal* cutoff = $C + U_y$
 - Else
 - Choose y
 - Generate *equal* cutoff = $L_x - C$
- Else if $T = \text{min}$
 - If $U_x \neq L_x$
 - Choose x
 - Generate *min* cutoff = $C + L_y$
 - Else
 - Choose y
 - Generate *max* cutoff = $U_x - C$
- Else (no cutoff)
 - If $U_x \neq L_x$
 - Choose x
 - Else
 - Choose y
- If T is *equal*, both a *max* and a *min* cutoff could be generated and passed down the tree. However, the search algorithm as implemented does not record two cutoff values, so a *max* cutoff is used arbitrarily.

x

• $f(x,y) = x \times y$

• If $x > 0$ and $y > 0$

```

    If  $T = \text{max}$ 
      If  $U_x \neq L_x$ 
        Choose  $x$ 
        Generate  $\text{max}$  cutoff =  $C/L_y$ 
      Else
        Choose  $y$ 
        Generate  $\text{min}$  cutoff =  $C/L_x$ 
    Else if  $T = \text{equal}$ 
      If  $U_x \neq L_x$ 
        Choose  $x$ 
        Generate  $\text{equal}$  cutoff =  $C/L_y$ 
      Else
        Choose  $y$ 
        Generate  $\text{equal}$  cutoff =  $C/L_x$ 
    Else if  $T = \text{min}$ 
      If  $U_x \neq L_x$ 
        Choose  $x$ 
        Generate  $\text{min}$  cutoff =  $C/U_y$ 
      Else
        Choose  $y$ 
        Generate  $\text{max}$  cutoff =  $C/U_x$ 
    Else (no cutoff)
      If  $U_x \neq L_x$ 
        Choose  $x$ 
      Else
        Choose  $y$ 

```

```

Else
  If  $U_x \neq L_x$ 
    Choose  $x$ 
  Else
    Choose  $y$ 

```

• The first test in the algorithm is necessary because the sense of the bounds and cutoffs changes as the signs of the inputs change. Only the positive-positive quadrant was implemented with cutoffs in Ulysses.

If T is *equal*, both a *max* and a *min* cutoff could be generated and passed down the tree. However, the search algorithm as implemented does not record two cutoff values, so a *max* cutoff is used arbitrarily.

Fn-2
Accel-eqn
Car-accel-eqn
Fn-x-sym
Fn-2-set
Make-tcd-class

- Function definitions: see Appendix D.
- Choose the first input $x: L_x \neq U_x$
- Note that for Accel-eqn and Car-accel-eqn, we could theoretically compute cutoff values. However, in practice this is not worthwhile because the robot speed is usually known, and the constraint speed and distance are determined at the same time. Make-tcd-class could also generate cutoffs if we examined the input combinations on a case-by-case basis.

Eq1

- $f(x, y) = (x = y)$
- If $L_x \neq U_x$
 - Choose y
 - If $L_y = U_y$
 - Generate *equal* cutoff = y
- Else
 - Choose x
 - If $L_x = U_x$
 - Generate *equal* cutoff = x

If-sym

- $f(p, x, y) = \begin{cases} x & p = \text{True} \\ y & p = \text{False} \end{cases}$
- If $L_p \neq U_p$
 - Choose p
- Else if $L_x \neq U_x$
 - Choose x
 - Generate T cutoff = C
- Else
 - Choose y
 - Generate T cutoff = C

Intersection

- $f(x_1, x_2, \dots) = x_1 \cap x_2 \cap \dots$
- If T is not null
 - Choose the first input $x_n: (L_{x_n} \neq U_{x_n}) \wedge (C \cap L_{x_n} \neq \emptyset)$
- Else
 - Choose the first input $x_n: L_{x_n} \neq U_{x_n}$

Member-p

- $f(x,y) = (x \in y)$
- If $L_x \neq U_x$
 - Choose x
- Else
 - Choose y
 - Generate *equal* cutoff = x

References

- [AASHTO 84] AASHTO.
A Policy on Geometric Design of Highways and Streets
American Association of State Highway and Transportation Officials,
Washington, D.C., 1984.
- [Aasman 88] Aasman, Jans.
Implementations of Car-Driver Behaviour and Psychological Risk Models.
In J. A. Rothengatter and R. A. deBruin (editors), *Road User Behaviour: Theory and Practice*, pages 106-118. Van Gorcum, Assen, 1988.
- [Agre 87] Agre, P. E. and D. Chapman.
Pengi: An Implementation of a Theory of Activity.
In *Proceedings of the Sixth National Conference on Artificial Intelligence*,
pages 268-272. Morgan Kaufman Publishers, Los Altos, 1987.
- [Akatsuka 87] Akatsuka, H. and I. Shinichiro.
Road Signposts Recognition System.
In *Proceedings of the SAE International Congress*. SAE, Detroit, 1987.
- [Aloimonos 88] Aloimonos, J., I. Weiss and A. Bandyopadhyay.
Active Vision.
International Journal of Computer Vision 1(4):333 - 356, 1988.
- [Ballard 91] Ballard, D. H.
Animate Vision.
Artificial Intelligence 48:57-86, 1991.
- [Belcher 89] Belcher, P. L. and Catling, I.
Autoguide Electronic Route Guidance in London and the U.K.
Technical Report 89102, ISATA, June, 1989.
- [Bender 69] Bender, J. G. and Fenton, R. E.
A Study of Automatic Car Following.
IEEE Transactions on Vehicular Technology VT-18:134 - 140, Nov., 1969.
- [Berliner 79] Berliner, H.
The B* Tree Search Algorithm: A Best-First Proof Procedure.
Artificial Intelligence 12:23-40, 1979.
- [Besl 85] Besl, P. J. and R. C. Jain.
Three-dimensional Object Recognition.
ACM Computing Surveys 17(1), March, 1985.
- [Binford 82] Binford, T. O.
Survey of Model-Based Image Analysis Systems.
International Journal of Robotics Research 1(1), 1982.
- [Blythe 89] Blythe, J. and Mitchell, T. M.
On Becoming Reactive.
In *Proceedings of the 6th International Workshop on Machine Learning*.
1989.
- [Brooks 82] Brooks, R.
Symbolic Error Analysis and Robot Planning.
AI Memo 685, MIT, September, 1982.

- [Burt 88] Burt, P.
'Smart Sensing' in Machine Vision.
Perspectives in Computing. Volume 20. *Machine Vision: Algorithms, Architectures, and Systems*.
In Herbert Freeman,
Academic Press, Inc., San Diego, CA, 1988, pages 1-30, Chapter 1.
- [Carbonell 91] Carbonell, J., C. Knoblock and S. Minton.
Prodigy: An Integrated Architecture for Planning and Learning.
Architectures for Intelligence.
In Kurt VanLehn,
Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1991, Chapter 9.
- [Cardew 70] Cardew, K. H. F.
The Automatic Steering of Vehicles -- An Experimental System Fitted to a Citroen Car.
Technical Report RL 340, Road Research Laboratory, February, 1970.
- [Chin 85] Chin, H.C.
SIMRO: A Model To Simulate Traffic at Roundabouts.
Traffic Engineering and Control 26(3):109 - 113, March, 1985.
- [Chrisman 91] Chrisman, L. and R. Simmons.
Sensible Planning: Focusing Perceptual Attention.
In *Proceedings of AAAI-91*. AAAI, 1991.
- [Clune 88] Clune, E.; Crisman, J.; Klinker, G.; and Webb, J.
Implementation and Performance of a Complex Vision System on a Systolic Array Machine.
Future Generations Computer Systems 4:15 - 29, 1988.
- [Crisman 88] Crisman, J. and C. Thorpe.
Color Vision for Road Following.
In *Proceedings of the SPIE Conference on Mobile Robots*. SPIE, 1988.
- [Cro 70] Cro, J. W. and Parker, R. H.
Automatic Headway Control -- An Automobile Vehicle Spacing System.
Technical Report 700086, Society of Automotive Engineers, January, 1970.
- [Crowley 85] Crowley, J.
Navigation for an Intelligent Mobile Robot.
IEEE Journal of Robotics and Automation RA-1(1), March, 1985.
- [Davis 87] Davis, E.
Constraint Propagation with Intervals.
Artificial Intelligence 32:281-331, 1987.
- [Dean 88] Dean, T. and Boddy, M.
An Analysis of Time-Dependent Planning.
In *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 49 - 54. Morgan Kaufmann, 1988.
- [Dickmanns 86] Dickmanns, E. and Zapp, A.
A Curvature-Based Scheme for Improving Road Vehicle Guidance by Computer Vision.
In *Mobile Robots (Vol. 727)*. SPIE, 1986.

- [Doyle 86] Doyle, R., D. Atkinson, and R. Doshi.
Generating Perception Requests and Expectations to Verify the Execution of Plans.
In *Proceedings of the Fifth National Conference on Artificial Intelligence*.
AAAI, 1986.
- [Drew 68] Drew, Donald.
Traffic Flow Theory and Control.
McGraw-Hill Book Company, New York, 1968.
- [Elliott 82] Elliott, R. J. and Lesk, M. E.
Route Finding in Street Maps by Computers and People.
In *Proceedings of AAAI 82*, pages 258 - 261. AAAI, 1982.
- [Ettinger 88] Ettinger, G. J.
Large Hierarchical Object Recognition Using Libraries of Parametrized Sub-parts.
In *Proceedings of the Conference on Computer Vision and Pattern Recognition*, pages 32 - 41. IEEE, Ann Arbor, Michigan, June, 1988.
- [Federal Highway Administration 78] FHWA.
Manual on Uniform Traffic Control Devices
Federal Highway Administration, U.S. Department of Transportation ,
Washington, D.C., 1978.
- [Federal Highway Administration 80] FHWA.
Traffic Network Analysis with NETSIM: A User Guide
Federal Highway Administration, Washington, D.C., 1980.
- [Fenton 91] Fenton, R. and Mayhan, R.
Automated Highway Studies at the Ohio State University -- An Overview.
IEEE Transactions on Vehicular Technology 40(1):100 - 113, February, 1991.
- [Fikes 72] Fikes, R; Hart, P; and Nilsson, N.
Learning and Executing Generalized Robot Plans.
Artificial Intelligence 3(4):251-288, 1972.
- [Firby 87] Firby, J. R.
An Investigation into Reactive Planning in Complex Domains.
In *Proceedings of the Sixth National Conference on Artificial Intelligence*,
pages 202-206. Morgan Kaufmann, 1987.
- [Firby 89] Firby, R. J.
Task Directed Sensing.
In *Vol. 1198, Sensor Fusion II: Human and Machine Strategies*, pages 480-489. SPIE, 1989.
- [Fujimori 88] Fujimori, T. and T. Kanade.
Knowledge-Based Interpretation of Outdoor Road Scenes.
In C. Thorpe and T. Kanade (editors), *CMU-RI-TR-88-4: 1987 Year End Report for Road Following at Carnegie Mellon*, pages 45-96. Carnegie Mellon University, 1988.
- [Gage 87] Gage, D. W. and Pletta, B. J.
Ground Vehicle Convoying.
In *Mobile Robots II (Vol. 852)*, pages 319 - 328. SPIE, 1987.

- [Gardels 60] Gardels, K.
Automatic Car Controls For Electronic Highways.
Technical Report GMR-276, General Motors Research Labs, Warren, MI,
June, 1960.
- [Georgeff 87] Georgeff, M. P. and Lansky, A. L.
Reactive Reasoning and Planning.
In *Proceedings of the Sixth National Conference on Artificial Intelligence*,
pages 677-682. Morgan Kaufman Publishers, Los Altos, 1987.
- [Gibson 81] Gibson, D. R. P.
Available Computer Models for Traffic Operations Analysis.
In *The Application of Traffic Simulation Models. TRB Special Report 194*,
pages 12 - 22. National Academy of Sciences, 1981.
- [Grimson 90] Grimson, W.E.L.
The Combinatorics of Object Recognition in Cluttered Environments Using
Constrained Search.
Artificial Intelligence 44:121-165, 1990.
- [Griswold 89] Griswold, N.C. and B. Bergenback.
Stop Sign Recognition for Autonomous Land Vehicles (ALVs) Using
Morphological Filters and Log-Conformal Mapping.
1989.
Texas A & M University, College Station, Texas.
- [Hanson 78] Hanson, A. and E. Riseman.
Segmentation of Natural Scenes.
Computer Vision Systems.
In A. Hanson and E. Riseman,
Academic Press, New York, 1978, pages 129 - 163.
- [Hayes-Roth 85] Hayes-Roth, B.
A Blackboard Architecture for Control.
Artificial Intelligence 26:251 - 321, 1985.
- [Hayes-Roth 90] Hayes-Roth, B.
Making Intelligent Systems Adaptive.
In K. Van Lehn (editor), *Architectures For Intelligence*. Erlbaum
Associates, 1990.
- [Hebert 88] Hebert, M. and T. Kanade.
3-D Vision for Outdoor Navigation by an Autonomous Vehicle.
In C. Thorpe and T. Kanade (editors), *CMU-RI-TR-88-4: 1987 Year End
Report for Road Following at Carnegie Mellon*, pages 29-41. Carnegie
Mellon University, 1988.
- [Horvitz 89] Horvitz, Eric J.
Reasoning about Beliefs and Actions under Computational Resource
Constraints.
In L.N. Kanal, T.S. Levitt, and J.F. Lemmer (editors), *Uncertainty in
Artificial Intelligence*, pages 301 - 324. Elsevier Science Publishers,
1989.
- [Huttenlocher 90] Huttenlocher, D. and S. Ullman.
Recognizing Solid Objects by Alignment with an Image.
International Journal of Computer Vision 5(2):195-212, 1990.

- [Ikeuchi 90] Ikeuchi, K. and M. Hebert.
Task Oriented Vision.
Proceedings of the DARPA 1990 Image Understanding Workshop :497 - 507, 1990.
- [Kaelbling 88] Kaelbling, Leslie P.
Goals as Parallel Program Specifications.
In *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 60 - 65. Morgan Kaufmann, 1988.
- [Kawashima 91] Kawashima, H.
Two Major Program Demonstrations in Japan.
IEEE Transactions on Vehicular Technology 40(1):141 - 146, February, 1991.
- [Kehtarnavaz ed] Kehtarnavaz, N., Lee, J. S., and Griswold, N. C.
Vision-Based Convoy Following by Recursive Filtering.
To be published .
- [Kluge 88] Kluge, K. and H. Kuga.
Car Recognition for the CMU Navlab.
In C. Thorpe and T. Kanade (editors), *CMU-RI-TR-88-4: 1987 Year End Report for Road Following at Carnegie Mellon*, pages 99-115. Carnegie Mellon University, 1988.
- [Kluge 89] Kluge, K. and C. Thorpe.
Explicit Models for Road Following.
In *Proceedings of the IEEE Conference on Robotics and Automation*. IEEE, 1989.
- [Kories 88] Kories, R., Rehfeld, N. and Zimmermann, G.
Toward Autonomous Convoy Driving: Recognizing the Starting Vehicle in Front.
In *Proceedings of the 9th International Conference on Pattern Recognition*, pages 531 - 535. IEEE, 1988.
- [Kuan 88] Kuan, D., Phipps, G. and Hsueh, A-C.
Autonomous Robotic Vehicle Road Following.
IEEE Transactions on PAMI 10(5):648 - 658, 1988.
- [Laird 87] Laird, John E; Newell, Allen; and Rosenbloom, Paul S.
Soar: an Architecture for General Intelligence.
Artificial Intelligence 33:1 - 64, 1987.
- [Laird 89] Laird, John E.; Yager, Eric S.; Tuck, Christopher M.; and Hucka, Michael.
Learning in Tele-autonomous Systems using Soar.
In *Proceedings of the 1989 NASA Conference on Space Robotics*. NASA, 1989.
- [Lang 79] Lang, R. P. and Focitag, D. B.
Programmable Digital Vehicle Control System.
IEEE Transactions on Vehicular Technology VT-28:80 - 87, Feb, 1979.
- [Lawler 66] Lawler, E. and D. Wood.
Branch-and-Bounds Methods: A Survey.
Operations Research 14:699-719, 1966.

- [Legislative Reference Bureau 87] Legislative Reference Bureau.
Pennsylvania Consolidated Statutes, Title 75: Vehicles (Vehicle Code)
Commonwealth of Pennsylvania, Harrisburg, PA, 1987.
- [Lin 89] Lin, Long-Ji; Simmons, Reid; and Fedor, Christopher.
Experience with a Task Control Architecture for Mobile Robots.
Technical Report CMU-RI-TR-89-29, Carnegie Mellon University Robotics
Institute, 1989.
- [Lowrie 85] Lowrie, James .
The Autonomous Land Vehicle (ALV) Preliminary Road-Following
Demonstration.
In David P. Casasent (editor), *SPIE Vol. 579, Intelligent Robots and
Computer Vision*, pages 336-350. SPIE, September, 1985.
- [Malec 91] Malec, J.
*Title? (required) [How to Pass an Intersection, or Automata Theory is Still
Useful.*
RKLLAB Research Report LiTH-IDA-R-91-08, Department of Computer
and Information Science, Linköping University, Linköping, Sweden,
1991.
- [Masaki 92] I. Masaki (editor).
Vision-based Vehicle Guidance.
Springer-Verlag, 1992.
- [McKeown 88] McKeown, D. M. Jr. and J. L. Denlinger.
Cooperative Methods For Road Tracking in Aerial Imagery.
In *Proceedings of the 1988 DARPA IUS Workshop*, pages 327-341. Morgan
Kaufmann, 1988.
- [McKnight 70] McKnight, J. and B. Adams.
Driver Education and Task Analysis Volume I: Task Descriptions.
Final Report, Department of Transportation, National Highway Safety
Bureau, Washington, D.C., November, 1970.
- [Michon 85] Michon, J. A.
A Critical View of Driver Behavior Models: What Do We Know, What
Should We Do?
In L. Evans and R. Schwing (editors), *Human Behavior and Traffic Safety.*
Plenum, 1985.
- [Minsky 75] Minsky, M.
A Framework for Representing Knowledge.
The Psychology of Computer Vision.
In Patrick Winston,
McGraw-Hill Book Company, New York, 1975.
- [Moravec 84] Moravec, Hans P.
Locomotion, Vision and Intelligence.
In Michael Brady and Richard Paul (editors), *Robotics Research*, pages 215
- 224. MIT Press, Cambridge, Massachusetts, 1984.

- [NadjmTehrani 91] Nadjm-Tehrani, S.
Analysis of the Overtaking Scenario: Specification of an Autonomous Car and a Driver Support System.
 RKLLAB Research Report LiTH-IDA-R-91-07, Department of Computer and Information Science, Linkoping University, Linkoping, Sweden, 1991.
- [Nitao 86] Nitao, John and Parodi, Alexandre M.
 A Real-Time Reflexive Pilot for an Autonomous Land Vehicle.
Control Systems Magazine 6(1):14 - 23, 1986.
- [Oshima 65] Oshima, R. *et al.*
 Control System for Automobile Driving.
 In *Proceedings of the Tokyo IFAC Symposium*, pages 347 - 357. , 1965.
- [Pomerleau 89] Pomerleau, D. A.
ALVINN: An Autonomous Land Vehicle In a Neural Network.
 Technical Report CMU-CS-89-107, Carnegie Mellon University, 1989.
- [Reece 88] Reece, D. A. and S. Shafer.
 An Overview of the Pharos Traffic Simulator.
 In J. A. Rothengatter and R. A. deBruin (editors), *Road User Behaviour: Theory and Practice*. Van Gorcum, Assen, 1988.
- [Reece 91] Reece, D. and S. Shafer.
A Computational Model of Driving for Autonomous Vehicles.
 Technical Report CMU-CS-91-122, Carnegie Mellon University, April, 1991.
- [Reid 80] Reid, L. D.; Graf, W. O.; and Billing, A. M.
The Validation of a Linear Driver Model.
 Technical Report 245, UTIAS, March, 1980.
- [Rillings 91] Rillings, J. H. and Betsold, R. J.
 Advanced Driver Information Systems.
IEEE Transactions on Vehicular Technology 40(1), February, 1991.
- [Rothengatter 88] T. Rothengatter and R. de Bruin (editor).
Road User Behaviour.
 Van Gorcum, 1988.
- [Sandewall 90] Sandewall, E.
Proposal for a ProArt Specification Platform.
 Technical Report LAIC-IDA-90-TR18, Department of Computer and Information Science, Linkoping University, Linkoping, Sweden, 1990.
- [Schoppers 87] Schoppers, M. J.
 Universal Plans for Reactive Robots in Unpredictable Environments.
 In *Proceedings of AAAI-87: Sixth National Conference on artificial Intelligence*, pages 1039 - 1046. Morgan Kaufman Publishers, Los Altos, 1987.
- [Shafer 86] Shafer, Steven A; Stenz, Anthony; and Thorpe, Charles E.
An Architecture for Sensor Fusion in a Mobile Robot.
 Technical Report CMU-RI-TR-86-9, CMU RI, April, 1986.
- [Shladover 91] Shladover, S. E. *et al.*
 Automatic Vehicle Control Developments in the PATH Program.
IEEE Transactions on Vehicular Technology 40(1):114 - 130, Feb, 1991.

- [Simmons 86] Simmons, R.
'Commonsense' Arithmetic Reasoning.
In *Proceedings of AAAI-86*. AAAI, 1986.
- [Solder 90] Solder, U. and V. Graefe.
Object Detection in Real Time.
In *Proceedings of the SPIE Symposium on Advances in Intelligent Systems*,
pages 121-165. SPIE, November, 1990.
- [Sugie 84] Sugie, M., Menzilcioglu, O., and Kung, H. T.
CARGuide -- On-board Computer for Automobile Route Guidance.
Technical Report CMU-CS-84-144, Carnegie Mellon University, 1984.
- [Texas Engineering Experiment Station 89]
Texas Engineering Experiment Station.
BART. Binocular Autonomous Research Team.
Research brochure, Texas A&M University.
1989
- [Thorpe 88] Thorpe, C., Hebert, M., Kanade, T., and Shafer, S.
Vision and Navigation for the Carnegie Mellon NAVLAB.
IEEE Transactions on PAMI 10(3), 1988.
- [Traffic Research Center 90]
Frank J. J. M. Steyvers (editor).
Annual Report 1989.
CIP-gegevens Koninklijke Bibliotheek, Den Haag, 1990.
- [Tsotsos 87] Tsotsos, J. K.
A 'Complexity Level' Analysis of Immediate Vision.
International Journal of Computer Vision 1(4):303 - 320, 1987.
- [Tsugawa 79] Tsugawa, S., Yatabe, T., Hirose, T., and Matsumoto, S.
An Automobile with Artificial Intelligence.
In *Proceedings of the 6th IJCAI*, pages 893 - 895. IJCAI, 1979.
- [Turk 87] Turk, M., Morgenthaler, D., Gremban, K., and Marra, M.
Video Road-Following for the Autonomous Land Vehicle.
In *Proceedings of the International Conference on Robotics and Automation*. IEEE, 1987.
- [Ullman 84] Ullman, S.
Visual Routines.
Cognition 18:97-160, 1984.
- [van der Molen 87]
van der Molen, H.H., and Botticher, A.M.T.
Risk Models for Traffic Participants: A Concerted Effort for Theoretical
Operationalizations.
Road Users and Traffic Safety.
In J. A. Rothengatter and R. A. de Bruin,
Van Gorcum, Assen/Maastricht, The Netherlands, 1987, pages 61-82.
- [von Tomkewitsch 91]
von Tomkewitsch, R.
Dynamic Route Guidance and Interactive Transport Management with
ALI-Scout.
IEEE Transactions on Vehicular Technology 40(1):45 - 50, February, 1991.

- [Waxman 87] Waxman, Allen; LeMoigne, Jacqueline J.; Davis, Larry S.; Srinivasan, Babu; Kushner, Todd R.; Liang, Eli and Siddalingaish, Tharakesh.
A Visual Navigation System for Autonomous Land Vehicles.
IEEE Journal of Robotics and Automation RA-3(2), April, 1987.
- [Wong 90] Wong, Shui-Ying.
TRAF-NETSIM: How It Works, What It Does.
ITE Journal 60(4):22 - 27, April, 1990.
- [Yarbus 67] Yarbus, A.
Eye Movements and Vision.
Plenum Press, New York, N.Y., 1967.
Original Russian text published by Nauka Press, Moscow, 1965.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admissions and employment on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders. In addition, Carnegie Mellon University does not discriminate in admissions and employment on the basis of religion, creed, ancestry, belief, age, veteran status or sexual orientation in violation of any federal, state, or local laws or executive orders. Inquiries concerning application of this policy should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.
